# Lightweight Modular Staging and Embedded Compilers: Abstraction without Regret for High-Level High-Performance Programming

THÈSE N$^{O}$ 5456 (2012)

PRÉSENTÉE LE 30 JUILLET 2012

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE MÉTHODES DE PROGRAMMATION 1

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Tiark ROMPF

acceptée sur proposition du jury:

Prof. C. Koch, président du jury
Prof. M. Odersky, directeur de thèse
Prof. V. Kuncak, rapporteur
Prof. K. Olukotun, rapporteur
Prof. W. Taha, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2012

# Abstract

Programs expressed in a high-level programming language need to be translated to a low-level machine dialect for execution. This translation is usually accomplished by a compiler, which is able to translate any legal program to equivalent low-level code. But for individual source programs, automatic translation does not always deliver good results: Software engineering practice demands generalization and abstraction, whereas high performance demands specialization and concretization. These goals are at odds, and compilers can only rarely translate expressive high-level programs to modern hardware platforms in a way that makes best use of the available resources.

Explicit program generation is a promising alternative to fully automatic translation. Instead of writing down the program and relying on a compiler for translation, developers write a program generator, which produces a specialized, efficient, low-level program as its output. However, developing high-quality program generators requires a very large effort that is often hard to amortize.

In this thesis, we propose a hybrid design: Integrate compilers into programs so that programs can take control of the translation process, but rely on libraries of common compiler functionality for help.

We present *Lightweight Modular Staging* (LMS), a generative programming approach that lowers the development effort significantly. LMS combines program generator logic with the generated code in a single program, using only types to distinguish the two stages of execution. Through extensive use of component technology, LMS makes a reusable and extensible compiler framework available at the library level, allowing programmers to tightly integrate domain-specific abstractions and optimizations into the generation process, with common generic optimizations provided by the framework. Compared to previous work on program generation, a key aspect of our design is the use of staging not only as a front-end, but also as a way to implement internal compiler passes and optimizations, many of which can be combined into powerful joint simplification passes.

LMS is well suited to develop embedded domain specific languages (DSLs) and has been used to develop powerful performance-oriented DSLs for demanding domains such as machine learning, with code generation for heterogeneous platforms including GPUs. LMS has also been used to generate SQL for embedded database queries and JavaScript for web applications.

**Keywords** Programming Languages, Compilers, Staging, Performance, Parallelism.

# Zusammenfassung

In einer “high-level” Programmiersprache geschriebene Programme müssen zur Ausführung in einen “low-level” Maschinendialekt übersetzt werden. Diese Übersetzung wird üblicherweise von einem Compiler durchgeführt, der jedes zulässige Programm in äquivalenten low-level Code übersetzen kann. Für individuelle Quellprogramme führt die automatische Übersetzung allerdings nicht immer zu einem guten Ergebnis: Anforderungen des Software-Engineering (Generalisierung und Abstrahierung) stehen im Widerspruch zu denen hoher Rechenleistung (Spezialisierung und Konkretisierung). Compiler können ausdrucksstarke Programme nur selten auf eine solche Weise übersetzen, dass sie die verfügbaren Ressourcen moderner Hardware-Plattformen in günstigster Weise ausnutzen.

Explizite Programmgenerierung is eine vielversprechende Alternative zur vollautomatischen Übersetzung. Anstatt ein Programm fertig auszuformulieren und sich auf einen Compiler zur Übersetzung zu verlassen, können Entwickler einen Programmgenerator schreiben. Dieser erzeugt ein spezialisiertes, effizientes low-level Programm als Ausgabe. Die Entwicklung qualitativ hochwertiger Programmgeneratoren erfordert allerdings einen sehr großen Aufwand der schwer zu amortisieren ist.

Diese Dissertation schlägt ein Hybrid-Modell vor: Man integriere Compiler in Programme so dass die Programme Kontrolle über den Übersetzungsvorgang übernehmen können wobei sie von Bibliotheken gängiger Übersetzerfunktionalität unterstützt werden.

Wir präsentieren *Lightweight Modular Staging* (LMS), eine Methode zur Programmgenerierung die den Entwicklungsaufwand deutlich verringert. LMS verbindet Logik des Programmgenerators mit erzeugtem Code im gleichen Programm und unterscheidet die zwei Ausführungsstufen anhand von Typen. Durch weitreichenden Einsatz von Komponenten macht LMS ein erweiterbares Compiler Framework als Bibliothek verfügbar. Dies erlaubt Prorammierern die enge Integration von Programm-spezifischen Abstraktionen und Optimierungen in den Generatorprozess, wobei generische Optimierungen vom Framework bereit gestellt werden. Ein wesentlicher Aspekt ist die Verwendung von Staging nicht allein als Front-End sondern auch zur implementierung interner Compiler-Stufen.

LMS ist gut geeignet zur Entwicklung von Domain-Specifc Languages (DSLs) und wurde erfolgreich eingesetzt um Performance-orientierte DSLs für anspruchsvolle Bereiche wie Machine Learning zu implementieren, einschließlich Code Generierung für GPUs. LMS wurde ebenfalls eingesetzt um SQL for eingebettete Datenbankabfragen und JavaScript für Web Anwendungen zu erzeugen.

**Schlagwörter** Programmiersprachen, Übersetzer, Staging, Performance, Parallelismus.

# Acknowledgments

I would like to thank my advisor Martin Odersky for giving me the opportunity to work on exciting projects, introducing me to the right people at the right times, and betting a large ERC grant proposal on my early ideas. I would also like to thank the members of my thesis jury, Christoph Koch, Viktor Kuncak, Kunle Olukotun and Walid Taha for their time and helpful comments. I am also grateful to my late MS advisor Walter Dosch for encouraging me to pursue a PhD in the first place.

The work presented in this thesis is part of a large collaborative effort driven by many people, both at EPFL and at Stanford University. It is hard to do justice to the profound and numerous ways this thesis was shaped by the joint work on Delite with Kevin Brown, Hassan Chafi, HyoukJoong Lee, Arvind Sujeeth, and Kunle Olukotun (Stanford). A huge thanks and it's a pleasure working with you! Additional thanks go to Adriaan Moors for the joint work on Scala-Virtualized, Christopher Vogt for SIQ, Vlad Ureche for StagedSAC, Philipp Haller and Manohar Jonalagedda for debugging and profiling facilities, Nada Amin and Grzegorz Kossakowski for embedded JavaScript, Stefan Ackermann, Vojin Jovanovic and Aleksandar Prokopec for distributed and collection operations (EPFL).

I would also like to thank everybody at LAMP, present and past, for providing a great and enjoyable work environment.

Last but not least, I would like to thank my friends and my family for their support and for providing a healthy counterbalance to work and research, in particular my parents and my lovely wife Maria.

# Contents

# Chapter 1

# Introduction

Computer programs are written by humans and executed by machines. But humans and machines do not speak the same language. Programs expressed in a *high-level* programming language convenient for human programmers need to be translated to a *low-level* machine dialect. This translation is usually accomplished by a separate program, a *compiler*. Compilers can translate any legal program to equivalent low-level code. But for individual source programs, automatic translation does not always deliver good results.

High-level programming languages such as Scala increase development productivity by focusing on abstraction and generalization. Efficient execution, by contrast, demands concretization and specialization. In many cases, programmers are forced to rewrite large parts of their own and possibly third-party code in a low-level style, which is time consuming and makes the program harder to understand and maintain. **How can we scale productivity gains to performance-critical software?**

On the hardware side, performance increasingly depends on multi-core processors and specialized accelerators such as GPUs, which are much harder to program efficiently than traditional sequential processors. The trend towards heterogeneous execution targets is not just performance driven, though. Web applications, for example, must deliver client components as executable JavaScript. Similar constraints apply to apps for mobile devices. **How can we target heterogeneous platforms from single high-level programs?**

In broad terms, these are the research questions this thesis seeks to address. The software industry is facing a difficult tradeoff: Hardware developments and heterogeneous targets pose increasing productivity challenges, and expressive programming languages pose growing performance challenges. Compilers have not kept pace for two primary reasons:

1. **Abstraction penalty:** High level programs contain a lot of indirection and redundant dispatch overhead due to module and abstraction boundaries which are essential for productivity. These indirections impose a runtime cost and perhaps more importantly, they cloud the compiler's view of the program so that it becomes much harder to apply optimizations.

2. **Lack of domain-specific optimizations:** To obtain good performance a compiler needs to optimize programs based on semantic knowledge about the program operations.

General purpose language compilers can only perform generic optimizations but cannot employ knowledge about the specific application domain (particularly important for efficient parallelization).

Other reasons include:

1. **Dependence on program input:** Domain specific optimizations or abstraction overhead may depend on parts of the program input, which is not available at compile time.

2. **Overwhelming number of choices:** An optimizing compiler is confronted with a large set of choices with uncertain outcomes. Exploring them all is not tractable for large programs.

An alternative to automatic program translation is explicit program generation. In this case, programmers write a high-level program that produces a lower-level program as its output. This approach has the potential to solve the issues listed above but comes with a set of other challenges:

1. **Not much support to build program generators**. Compilers are well understood. The development effort for a new (general purpose) compiler is large but it can be amortized over a huge number of programs. Program generators are less widespread and more specialized. There is much less practical experience to draw from. The development effort is also large but the scope of use is much narrower. Thus, building program generators is only feasible for key libraries (FFT, BLAS), not for everyday programming. Specialized *multi-stage* programming languages[133] aim to increase productivity in building code generators by providing special syntax that allows to mark *staged* program expressions that will become part of the generated program (as opposed to the generator). While there are certain well-formedness and type safety guarantees, the program abstraction is still essentially syntactic, and programmers have to explicitly manage the execution order of staged expressions. This makes it hard to avoid subtle errors caused by duplicating or reordering computation. Moreover, since the original intention is to generate low-level programs, it is often desirable to have a more restricted target language.

2. **Lack of generic optimizations**. Program generators can easily implement domain specific optimizations that are expressible as *specializations* of a certain template. But to work well in a more general setting, they *also* need to apply all the generic optimizations of a regular compiler. Generic and specific optimizations need to be integrated for maximum effectiveness. This means that common compiler optimizations that, for example, remove redundant or unnecessary computations or move computation out of loops, need to be applied on the level of domain operations, for example, on matrix and vector operations in a linear algebra program. Applying these common optimizations only on the level of generated low-level code is too late. For example, a duplicated matrix multiplication will lead to two sets of nested loops that a low-level compiler cannot easily recognize as identical. This is a key reason why compiler preprocessors and macro systems are only suited for a limited range of optimizations.

## 1.1 Embedded Compilers: A Hybrid Approach

In this dissertation we investigate an alternative approach to fully automatic translation or fully manual program generation. We propose a hybrid design: integrate compilers into programs so that programs can take control of the translation process, but rely on libraries of common compiler functionality for help. This raises three important questions. If compilers are embedded into programs then:

1. **What language does an embedded compiler understand?** Compared to general-purpose languages, embedded compilers understand simpler but larger languages. Their languages are domain specific: they provide only a minimum of abstraction facilities but lots of concrete operations with efficient translations. Embedded compilers are essentially libraries. They are modular and extensible through modularity features of the language they are written in. For the embedded languages, core constructs and generic optimizations are provided by a framework. New domain specific features and optimizations can be added by additional libraries or by the application itself. Depending on the code generation target, compiler modules exist that lift large parts of the standard library of the enclosing language to the embedded domain and add library specific optimizations.

2. **Which programs does an embedded compiler translate?** Embedded compilers do not implement lexers, parsers or type checkers. They rely on the surrounding program to provide input in an intermediate representation (IR). Embedded compilers expose the syntax of their language as an API that the program can use to create and assemble IR expressions. This API behaves just like a library that would otherwise directly execute the operations, but instead it creates IR nodes. In this thesis, expressions that create IR nodes are identified by their type. Many operations on regular values are lifted to the domain of IR nodes. The types of the arguments define which version is used, whether the operation is executed directly or an IR node created. Compiler API calls and direct computation can be mixed freely in the same program. This way, the program can resolve indirection and abstraction overhead through immediate computation and will create IR expressions only for those operations that need to run fast.

3. **What is the output of an embedded compiler?** The final output of an embedded compiler is low-level source or binary code in one or more target languages (e.g. Scala, C, assembly, CUDA, JavaScript, SQL) that the program can load and run at its convenience. Embedded compilers can also be layered to form multi-pass compilation pipelines. In that case, the program traverses (i.e. interprets) the IR-form output of one compiler and computes input for another, possibly lower-level compiler. Thus, program transformation passes can take advantage of the same benefits as the initial program generation. Most importantly, they need not implement low-level IR manipulation but use the regular compiler API, intermixed with direct computation to resolve abstraction.

The rest of this thesis will explain the technical details and demonstrate the validity and usefulness of this approach.

## 1.2 Lightweight Modular Staging

Embedded compilers try to overcome the performance and productivity gap by leveraging productivity where performance does not matter and vice versa: Generating embedded programs and compiling them is not performance critical but productivity is paramount to make the development of staged programs and embedded compilers affordable. Embedded programs need to run fast but they are generated programmatically and not written by humans, so programmer productivity is not a concern.

Programming with embedded compilers makes a pattern explicit that is found in many programs, namely a division into computational *stages*, separated by frequency of execution or availability of information [68]. Thus, embedded compilers can be seen as a form of *multi-stage programming (MSP)* [130]. We will refer to the program phase of constructing embedded program IR nodes via the embedded compiler API as *staging time*. The particular method, developed throughout this thesis, is called *lightweight modular staging (LMS)* [110, 111]. Compared to previous multi-stage programming approaches, LMS uses types to identify stages instead of syntax annotations, provides execution order guarantees, and offers a principled interface for extensible optimizations.

## 1.3 Combining Staging and Extensible Compilers

In order to reason about user or library defined abstractions compilers need to be extensible. Common mechanisms to extend compilers fall into two categories. Frontend macros, staging or partial evaluation systems can be used to programmatically remove abstraction and specialize programs before they enter the compiler. Alternatively, some compilers allow extending the internal workings by adding new transformation passes at different points in the compile chain or adding new IR types. None of these mechanisms alone is sufficient to handle the challenges posed by high level abstractions such as domain specific data structures. We present a novel way to combine them to yield benefits that are greater than the sum of the parts.

Instead of using staging merely as a front end, we implement internal compiler passes using staging as well. These internal passes delegate back to program execution to construct the transformed IR. Staging is known to simplify program generation, and in the same way it can simplify program transformation. Defining a transformation as a staged IR interpreter is simpler than implementing a low-level IR to IR transformer. Custom IR nodes can be added easily and many optimizations that are expressed as rewritings from IR nodes to staged program fragments can be combined into a single pass, mitigating phase ordering problems. Speculative rewriting can preserve optimistic assumptions around loops.

## 1.4 Deep Linguistic Reuse

Similar to previous MSP approaches, LMS leverages linguistic reuse [80] to increase productivity for developing program generators by expressing program generator and generated

code in a single program. We distinguish mostly syntactic *shallow* reuse from *deep* reuse that includes a nontrivial translation step: We say that we are reusing a language feature of the surrounding language in a deep way in embedded programs if the embedded compiler and its language do not need to implement the particular feature. Reusing the type system of the embedding language to produce type-correct target code without implementing type checking in an embedded compiler is an example of deep reuse. Deep reuse is key for productivity as it drastically simplifies embedded compiler development.

## 1.5 Abstraction Without Regret

As explained earlier, abstraction and generalization enable productivity but hinder performance by incurring interpretive overhead at runtime and obscuring semantic information at compile time, which complicates static analysis.

Deep reuse of abstraction facilities like functions or objects enables "abstraction without regret": Programmers can use arbitrary language features to structure their programs in the generator stage, with the comforting knowledge that these abstractions are guaranteed to be removed at staging time. No runtime price will need to be paid, and the embedded compiler will see a program that has the abstraction stripped away. Thus, deep reuse is also important for performance. We demonstrate deep reuse of a variety of language features including functions, continuations, type classes, and staging-time data structures.

## 1.6 Language Virtualization

Coupled with an expressive general purpose language, embedded compilers are a key component to enable what we call *language virtualization* [19]: Tweaking the language so that other languages can be embedded in it and embedded programs are "as good as" stand-alone programs. Of course "as good as" has many facets (expressiveness, safety, performance). Embedded compilers enable the performance aspect, which is otherwise a key obstacle for providing domain specific abstractions as libraries as opposed to building full fledged external domain specific languages.

This thesis presents Scala-Virtualized, an effort in virtualizing the Scala programming language. We understand virtualization also in the sense of virtual (i.e. overridable) methods, making core language features overridable. Overriding control structures such as conditionals is necessary to provide variants that create IR nodes for embedded compilers.

## 1.7 Domain-Specific Languages (DSLs)

With embedded compilers, developers can program with efficiently compiled domain-specific abstractions without having to build a complete DSL and a separate compiler toolchain. In other words, embedded compilers significantly enhance the power of the embedded DSL approach [58] of implementing DSLs as domain-specific libraries in an expressive general purpose host language.

Programmers get most of the benefits of a full external DSL (domain specific abstractions, efficient compilation; minus domain specific syntax) without the drawbacks (DSL development effort, overhead of several DSLs in a single program). They also get the benefits of an embedded DSL or a domain-specific library (familiar constructs across different domain-specific extensions: loops, functions, etc). Many host language libraries can be lifted to the compiled DSL domain, so that DSLs need not implement standard library functionality. DSLs can be built from components and interoperate through the host language.

## 1.8 Delite

We have implemented the concepts described in this thesis as part of the LMS-Core and Delite frameworks, which are the basis for a number of highly efficient embedded DSLs [126, 11, 113, 127, 83]. Delite is a parallelization and heterogeneous computing platform that started out as a library-based deferred execution runtime [18]. Version 2 of Delite uses LMS and embedded compilation with very good results to provide a full-stack embedded DSL framework.

Among the DSLs developed using Delite are the following:

- OptiML / OptiLA: Machine Learning and linear algebra
- OptiQL / OptiCollections: Collection and query operations
- OptiMesh: Mesh-based PDE solvers
- OptiGraph: Graph analysis

More details about Delite and the individual DSLs are presented in Chapter 16.

## 1.9 Contributions

In addition to demonstrating the power and usefulness of LMS and embedded compilers on the whole this thesis makes the following individual contributions:

**Staging**

- We present staging combinators that maintain the relative execution order of expressions within a stage. In previous work, arbitrary and context free combination of code fragments (plain strings or quasi-quoted expressions) may reorder effects or slow down programs by duplicating computation. We build on previous work on let-insertion, and on partial evaluation with side effects [137].
- We implement staging facilities as a library, with only minimal language support (virtualized control structures) and without extra syntax. Types within the language distinguish stages, and type inference provides a semi-automatic local binding-time analysis (BTA). Thus, the approach shares features of automatic partial evaluation and manual staging.

Previous work focused on dedicated MSP languages with explicit quasi-quote syntax (Lisp, MetaML), or on fully automatic partial evaluation. We build on previous work on embedding typed languages [55, 15].

- We combine staging with extensible compilation. In particular, staged IR interpreters act as IR transformers. Thus we show that staging is not only beneficial for program generation but also for program transformation. Among the previously presented MSP languages, some (but not all) allow inspection of staged ASTs but in general there are no facilities to integrate domain specific optimizations into the compilation pipeline of staged expressions.
- We present applications and case studies that use advanced host language abstractions as part of code generator and obtain the effect of those abstractions in generated code without overhead (abstraction without regret) and without much additional work (deep linguistic reuse). In particular we discuss closures, continuations, type classes, traits/objects, gradual typing, object literals, for comprehensions and pattern matching. Many staged applications presented previously used monadic front-ends, or explicit interpreters that were staged to remove overhead.

**Compiler Architecture**

- We present an extensible compiler architecture that allows to compose many independent optimizations and maintains optimistic assumptions for combined optimizations through speculative rewriting. In previous work, extensible compilers are often restricted to adding new phases and implementing combined optimizations often requires manually devising complicated superanalyses. We build on previous work on combining data flow analyses [85].
- We present a structured "sea of nodes" IR with nesting, frequency and parallelism information. On this IR we implement a code motion algorithm that can rearrange structured expressions such as nested loops or lambdas. The algorithm is driven by nesting instead of dataflow information. Most previously presented code motion algorithms rely an dataflow information which precludes their use for situations where flow information is not available.
- We present a novel data structure traversal and loop fusion algorithm that handles high-level data parallel loops and also supports asymmetric traversal operations, including `flatMap` and `groupBy`. In previous work, imperative loop fusion has been studied extensively but most algorithms only support flat array traversals, no `flatMap`, `filter`, or `groupBy`. In the functional setting, deforestation [153] and stream fusion [30] are able to combine linear traversals but neither data parallel loops nor `groupBy`.

**Artifacts** (developed in collaboration with others)

- Scala-Virtualized (Chapter 5): An extension of the Scala language to make embeddings more seamless.

- LMS-Core framework (Chapter 9): Common building blocks for core language features, lifting parts of the Scala standard library, generic optimizations.
- Delite DSL framework (Chapter 16): Data parallel operations and code generation for heterogeneous parallel devices.
- Various DSLs (Section 16.4, Chapter 17)

## 1.10 Terminology

Since much of the implementation work has been carried out in the context of compiled embedded DSLs that share common generic functionality, we will use the term "DSL" frequently with a broad meaning, for example to refer to arbitrary library modules that can construct program fragments for an embedded compiler (domain specific or not). Drawing a clear distinction between embedded DSLs and regular libraries seems like a futile undertaking in general. Moreover, we will use the following terms interchangeably (per line) except where otherwise noted:

| | | |
|---|---|---|
| host language | meta language | surrounding language |
| embedded language | object language | DSL |
| embedded program | object program | DSL program |
| embedded compiler | DSL compiler | |

If further precision is required, we take the term *object program* to denote a program expressed in an embedded compiler's IR representation and *DSL program* to denote the larger class of programs that may also contain operations that are evaluated at staging time. Likewise, *object language* refers to the language of an embedded compiler whereas *DSL* is the user-facing language that may contain staging-time operations.

## 1.11 Outline

Chapter 2 gives background information on hardware developments (Section 2.2), programming language advances (Section 2.4) and their respective challenges. Section 2.3 discusses domain-specific languages. Section 2.5 presents background on staging and generative programming.

Then two technical parts follow. The question Part I seeks to answer is: How do we define programs for embedded compilers? Correspondingly, Part II seeks to answer: How do embedded compilers compile these programs?

Within Part I, Chapter 3 reviews multi-stage programming as a way of increasing productivity in program generator development by linguistic reuse, namely embedding generated code templates and generation logic in a single program. Chapter 4 discusses language virtualization. Chapter 5 introduces Scala-Virtualized. Chapter 6 discusses staging from the point

of view of linguistic reuse, Section 6.1 reviews the state of the art and Section 6.2 presents the contributions of LMS, as far as they apply to generating concrete syntax from staged expressions.

Within Part II, Chapter 7 switches to the intermediate representation level and gives an overview of the challenges that embedded compilers face and the design decisions taken in this thesis. Chapter 8 discusses a straightforward intermediate representation based on expression trees. Chapter 9 presents a more refined, graph-based representation that is better suited to extreme modularity. Chapter 10 discusses advanced optimizations.

Part III demonstrates staging and embedded compilers at work, discussing small but in-depth examples, in particular how embedded compilers provide abstraction without regret. Chapter 11 presents an overview, Chapter 12 focuses on control abstraction and Chapter 13 on data abstraction. Chapter 14 discusses case studies related to data representation conversion and loop fusion, as well as more common staging challenges.

Part IV provides additional validation and evaluation by describing larger projects that use the techniques developed in this thesis. Chapter 16 discusses the Delite DSL Framework. Section 16.4 discusses individual Delite DSLs. Chapter 17 discusses other projects (StagedSAC, SIQ, Jet embedded JavaScript). Chapter 18 finishes by surveying related work.

# Chapter 2

# Background

This chapters expands upon some of the topics from the introduction, such as hardware trends, programming language trends, domain specific languages and generative programming. The discussion focuses on the prior state of the art, not including the contributions of this thesis.

## 2.1 Economics of Productivity, Performance and Safety

Software development is largely driven by economic concerns. Project managers and developers face a dilemma in choosing their tools: On the one hand, it is desirable to write very high level code and use programming abstractions close to the problem domain, while on the other hand, software is required to be efficient.

High-level abstractions make code easy to write and modify, thus economizing on development and maintenance hours. At the same time, high-level code is easy to understand for others. It is easier to reuse and adapt for new projects, so development efforts are more likely to amortize in the long run. In addition, the closer executable code is to a formal or informal specification, the easier it is to verify and test, reducing the chance of costly bugs in the field. Efficiency comprises low resource demands, so the system can be deployed on inexpensive hardware, as well as the system itself achieving low latency and high throughput.

Manually optimizing programs for performance incurs a great loss in productivity. Programmers typically start with a high-level implementation and translate it to low-level, high performance code in a time-consuming process. This optimization process obfuscates the programmer's intent and makes the code harder to understand, debug and maintain. The low-level code is tied to hardware details (e.g. cache-line size, number of processor cores) and is not easily portable to different architectures. A specification change may require re-optimizing the high-level code, or worse, the optimized code is modified and the high-level reference implementation is no longer in accordance with the specification. The use of low-level programming models is also detrimental to program safety because low-level code is more likely to attract bugs and is harder to verify correct. From a safety point of view, code generation and program synthesis are preferable to low-level post-hoc verification.

In other cases obtaining peak performance is not the primary objective but developers

need to cope with a heterogeneous programming environment due to external reasons. Interfacing with relational databases, for example, happens through SQL statements. Web applications contain server and client functionality where the client part that is supposed to run inside a web browser has to be provided as JavaScript code. This split makes it hard to refactor programs and move pieces of functionality from the server to the client or vice versa. Thus, equivalent functionality is often duplicated. In the case of web applications, input validation logic is frequently implemented twice, to be executed on the client for responsiveness and on the server for security. Both versions must be maintained independently and if validation logic is accidentally changed in only one place there is a potential security risk.

## 2.2 Hardware Trends and Productivity Challenges

### 2.2.1 Hitting the Power Wall

Power constraints have limited the ability of microprocessor vendors to scale single-core performance with each new generation. Instead, vendors are increasing the number of processor cores and incorporating specialized hardware such as GPUs and SIMD units to improve performance [61].

### 2.2.2 Proliferation of Programming Models

To take advantage of heterogeneous systems, programmers need expertise in a variety of programming models (Pthreads or OpenMP for multi-core CPU, OpenCL or CUDA for GPU, MPI for clusters [136, 96]). An efficient mapping of an application to a heterogeneous architecture needs to match the characteristics of the application to the different capabilities of the hardware. At this time, there are no compilers or other automatic solutions that can reliably create efficient mappings for entire programs. This suggests that for each application and for each computing platform a specialized mapping must be created by a programmer that is an expert in the specific domain as well as in the targeted parallel architecture. This approach is unsatisfactory and likely infeasible at scale, as it incurs an explosion and fragmentation of mapping solutions. The key challenge is to find ways to take good mapping solutions created by experts and reuse them for larger classes of programs or entire *application domains*.

## 2.3 Domain-Specific Languages (DSLs)

One way to capture application-specific knowledge for a whole class of applications and simplify application development at the same time is to use a domain specific language (DSL). A DSL is a concise programming language with a syntax that is designed to naturally express the semantics of a narrow problem domain [143]. Examples of commonly used DSLs are TeX and LaTeX for typesetting academic papers, Matlab for prototyping numerical linear algebra algorithms, and SQL for querying relational databases.

### 2.3.1 DSLs for Performance

DSLs have a long history of increasing programmer productivity by providing extremely high-level, in a sense "ideal", abstractions tailored to a particular domain. Performance-oriented DSLs strive to also make the *compiler* more productive (producing better code) by enabling it to reason on a higher level as well. While productivity and performance are often at odds in general-purpose languages, DSLs can expose significantly more semantic information about the application in addition to providing a means of writing concise, maintainable code. This semantic information, obtained in particular through domain constructs that expose structured, coarse-grained parallelism, makes it feasible for a DSL compiler to generate high performance code targeting parallel heterogeneous architectures, from high-level, single-source application code [18].

### 2.3.2 External DSLs

Despite the benefits of performance oriented DSLs, there are significant challenges to their widespread adoption. The dominant approach is to implement a DSL compiler tool chain from scratch. First, the cost of developing a DSL to a sufficient degree of maturity is immense and it is unclear whether this investment can be be made time and time again for all possible domains. This means that many DSLs will likely be immature and buggy. They will not provide state of the art tooling (IDEs, debuggers, profilers), they will be lacking in terms of modularity, and they are unlikely to support advanced programming constructs like objects or higher order functions due to their restricted execution model. Tools like Spoofax [71] address some of these issues by providing infrastructure for not only defining DSLs but also associated components like IDE editors and debuggers. There is also a tendency for DSLs to accumulate general purpose features over time but the implementation of those features will be substandard compared to general purpose languages ("inner platform effect"). Second, significant applications may need to use several DSLs. Having to use multiple different, incompatible, and immature DSLs to get high performance is not a clear advantage over the current low-level programming approach with multiple frameworks.

### 2.3.3 Embedded DSLs

Embedded, or internal DSLs require significantly less investment for development and use, especially if they are well integrated with the host language and environment [58, 59]. Such DSLs can be used like libraries and DSL programs can use regular host language syntax. Since embedded DSL programs can interoperate more easily with one another and with generic code written in the host language, programmers do not need to learn different syntax for common features and much of the host language tooling can be reused. However, embedded DSLs usually cannot perform optimizing transformations or compile to platforms not supported by the host language. Therefore, they are limited in the performance they can achieve on modern hardware. Individual compiled embedded DSLs have been studied [42, 84] but no general frameworks for their construction have emerged yet.

## 2.4 Programming Language Trends and Performance Challenges

General-purpose languages focus on primitives for abstraction and composition, so that programmers can build large systems from few and relatively simple but versatile parts. Modern expressive languages allow to build domain-specific language extensions as libraries (in fact, purely embedded DSLs). However, the compiler has no knowledge about the extended semantics.

### 2.4.1 The Abstraction Penalty

High-level programming languages such as Scala increase development productivity by focusing on abstraction and generalization. In fact, building and managing complex software systems is only possible by generalizing functionality and abstracting from particular use cases. But abstraction does not come for free. Efficient execution, by contrast, demands concretization and specialization. These requirements are at odds. High level programs contain a lot of indirection and redundant dispatch overhead due to module and abstraction boundaries which are essential for productivity.

These indirections incur interpretive overhead and thus impose a runtime cost. Perhaps more importantly, they also cloud the compiler's view of the program so that it becomes much harder to apply optimizations. Control flow analysis is a prerequisite for many other optimizations, but higher order control flow analysis is a tough problem because the flow of control depends on the flow of data, which again depends on the flow of control [145, 38].

### 2.4.2 The General Purpose Bottleneck

To obtain good performance a compiler needs to optimize programs based on semantic knowledge about the program operations. General purpose language compilers can only perform generic optimizations but cannot employ knowledge about the specific application domain. Domain knowledge is particularly important for efficient parallelization.

General-purpose compilers do not understand the semantics of the complex operations performed within an application. Reasoning across domain constructs, however, enables more powerful and aggressive optimizations that are infeasible otherwise. Because of the general-purpose nature needed to support a wide range of applications, compilers can usually infer little about the structure of the data or the nature of the algorithms the code is using. By contrast, DSL compilers can use aggressive optimization techniques using knowledge of the data structures and algorithms derived from the DSL.

General-purpose languages impose very few restrictions on programmers which in turn requires the compiler to perform non-trivial analyses to prove that optimizations are safe. Unfortunately, safety of optimizations often cannot be determined and therefore the compiler must be conservative and not apply the optimization to guarantee correctness of the generated code.

Performance-oriented DSLs can take the opposite approach to the problem, namely restrict the programmer from writing code that would prevent the compiler from generating

an efficient implementation. The compiler is then able to perform very aggressive optimizations with much simpler or even without safety analyses, providing the programmer with efficient code for significantly less effort. For example, a program requiring graph-analysis would express graph traversals using a breadth-first search or depth-first search language construct without spelling out the implementation of such traversals [57]. A corresponding domain-specific compiler can reason about the program at the level of domain operations, enabling coarse grain optimizations, for example eliminating whole linear algebra operations as opposed to individual arithmetic instructions. Furthermore, these abstractions leave implementation details unspecified, providing the compiler with the freedom to translate the application to a number of different low-level programming models.

### 2.4.3 Static or Dynamic Compilation

Domain specific optimizations or abstraction overhead may depend on parts of the program input, which is not available at compile time.

Programs can be compiled either ahead of time or just in time (JIT). Some modern language environments such as the Java Virtual Machine (JVM) employ a combination of these techniques. Virtual machines that employ JIT compilation obtain some information on dynamic behavior of the program, for example by call-site specific profiling or class analysis (HotSpot/V8). Nevertheless, compilation happens mostly automatically and it is hard for JIT compilers to guess the right specialization points. While VMs employ aggressive inlining they do not usually generate multiple specialized code paths based on program data (monovariant specialization only).

### 2.4.4 Combinatorial Explosion

An optimizing compiler is confronted with an overwhelmingly large set of choices with uncertain outcomes. It cannot possibly explore all of them and must rely on heuristics which, by their very nature, work some but not all of the time.

### 2.4.5 Functional Programming Challenges

Functional programming presents lots of theoretical opportunities for program optimization due to referential transparency: Evaluating a particular expression produces always the same result, independent of the context. Expressions can be reordered or eliminated as long as input dependencies are maintained. Control and anti-dependences do not exist in pure functional programs. This simplicity is also exploited by mostly-functional intermediate representations such as SSA-form [3].

In practice however, functional programs are often less efficient than tight imperative code because they create lots of temporary data structures instead of modifying data in place. In a sense, impure functional languages that discourage but do not disallow side effects get the worst of both worlds because their compilers cannot safely apply optimizations that rely on

functional properties without proving the absence of side effects with the help of sophisticated effect systems.

### 2.4.6 Scala Challenges

The Scalac compiler does not employ effect analysis and thus has to be very conservative, assuming worst case side effects in many places. This slows down functional programs in Scala noticeably. If generic types are instantiated to primitives, primitive values will need to be "boxed" in order to maintain the uniform representation as heap allocated objects. Boxing is expensive in terms of memory use, allocation cost and garbage collection overhead. Thus, programmers can request to specialize generics for primitive types by adding `@specialized` annotations to type parameters of classes or methods [36]. However, programmers must take care that indeed the full call paths are specialized, otherwise boxing will still occur. There is also a risk of code bloat: When naively specializing a class with three type parameters for all 10 primitive types, 1000 versions of this class will be produced.

The Scala compiler is also prone to a few instances of phase ordering problems. Early on, for example, local variables that are accessed by closures will be converted to heap allocated reference cells. Later, the closure allocation may get eliminated and the code inlined. However, the local variables have already been converted to heap objects and will not be converted back. Another example is tail call elimination happening before inlining. If an invocation becomes a recursive tail call only via inlining it will not be converted to a jump. Inlining in general has to be conservative, since Scalac has to operate under a fundamental open-world assumption. Inlining implicitly assumes that the inlined bytecode will not change between compilation and runtime. Scalac chooses to inline only (effectively) final methods that can never be overridden. This decision is slightly more conservative than it needs to be: If the exact type of an object is known at compile time even non-final methods could safely be inlined.

### 2.4.7 JVM Challenges

The Java Virtual Machine (JVM) comes with its own set of challenges. The uniform reference object representation that requires boxing of primitives was already mentioned above. A related problem is that there is no notion of structs or records that can be allocated on the stack or stored consecutively in arrays. Array accesses also require bounds checks. The JIT compiler will eliminate bounds checks in many cases but these mechanisms do not work reliably for complex access patterns. Furthermore, all freshly allocated arrays are initialized with zero values, which can impose some slowdown and hinder efficient parallelization.

The HotSpot JIT compiler performs aggressive inlining of "hot" methods driven by runtime profiling. The profiling is done on a per call site basis, which produces very good results if call sites are monomorphic, i.e. always call the same target method. However the scheme breaks down for higher order control flow [22]. Taking the higher order function `foreach` as an example, the number of distinct functions called from `foreach` is very large and thus every iteration step pays the overhead of a virtual method call. A much better strategy would be to

first inline `foreach` at its caller which would produce a monomorphic call to the particular closure argument.

## 2.5 Staging and Generative Programming

Generative programming is a promising alternative to fully automatic translation. Instead of writing down the target program directly, developers write a high-level, generic program generator, which produces a specialized, efficient program as its output. However, developing high-quality program generators requires a very large effort that is often hard to amortize.

Generative programming can be broadly classified as static or dynamic. Static code generation happens at compile time, a widely used example is the C++ template language or macro systems in other languages. Dynamic code generation that takes place at program runtime brings additional flexibility because code can be specialized with respect to parameters not yet available at compile time.

### 2.5.1 Multi-Stage Programming With Explicit Annotations

Many computations can naturally be separated into stages distinguished by frequency of execution or availability of information. Staging transformations aim at executing certain pieces of code less often or at a time where performance is less critical. In the context of program generation, multi-stage programming (MSP, *staging* for short) as established by Taha and Sheard [133] allows programmers to explicitly delay evaluation of a program expression to a later stage (thus, *staging* an expression). The present stage effectively acts as a code generator that composes (and possibly executes) the program of the next stage. A nice property of this approach is that generator and generated code are expressed in a single program, with the aim that programmers can construct a multi-stage program from a naive implementation of the same algorithm by adding staging annotations in a selective way.

Basic mechanisms for composing program fragments at runtime have existed for a long time in the form of Lisp's "code is data" model and its use of *quasi-quotation*: syntactic annotations to denote expressions that should remain unevaluated and to mark holes within them, to be filled in with expressions computed elsewhere. Dedicated MSP languages such as MetaML [133] and MetaOCaml [13] add well-scoping and well-typing guarantees about the generated code. Despite these advances, building "active" libraries or domain-specific languages (DSLs) that incorporate dynamic code generation remains a significant challenge.

### 2.5.2 Partial Evaluation

Partial evaluation is an automatic program transform that essentially performs very aggressive constant propagation [65]: Given some *static* parts of the program input, partial evaluation produces a *residual* program that is specialized to the already provided static inputs. Passing the remaining *dynamic* input to the residual program will produce the final result, but in a more efficient way than just executing the original program.

Partial evaluation literature calls stages "binding times". Where staging requires the programmer to define the binding times of each expression, partial evaluation determines the binding times automatically. Offline partial evaluation itself works in two stages, with a dedicated binding time analysis (BTA) followed by a specialization pass. BTA annotates each expression of the input program whether it can be computed statically or must be *residualized*, i.e. become part of the residual staged program. Staging can be seen as a manual form of BTA and BTA as a form of automatic staging.

Since partial evaluation is a form of program specialization, it usually comes with soundness guarantees and preserves semantics of programs. In general this is not the case with staging, where programmers can more freely compose program fragments. Nevertheless staging is often used with the intent of achieving specialization, so tighter semantic guarantees seem highly desirable.

An alternative to offline partial evaluation is online partial evaluation, which makes decisions what to specialize during the specialization process itself, instead of relying on a separate BTA. Hybrid approaches exist as well [128, 116]. Closely related to partial evaluation is supercompilation [141, 123], and there have been efforts to unify different notions of automatic program transformation [12, 122].

### 2.5.3 Generative Front-Ends and Extensible Compilers

Macro systems, staging and partial evaluation are used to specialize programs before they reach the real compiler. Thus, they provide a simple way of extending a general purpose language with domain-specific abstractions that can be resolved in a controlled way.

Another option is to teach the compiler itself new domain-specific rules [97, 41, 67]. In many cases, compiler extensibility is understood as allowing to add new phases. This leads to phase ordering issues if there are many optimization passes. Also, passes need to be defined as transformations on some kind of program IR. This is much more difficult than the macro or staging approach, which can use regular (multi-stage) computation to express the desired target code. Some extensible compilers also allow adding new IR types but often it is not clear how new nodes interact with existing generic optimizations.

We argue that none of these approaches alone is sufficient and a lot is to be gained if they are combined. To give an example, if all we have is staging or macros, and we implement a linear algebra package, then a matrix multiplication `m*id` will be expanded into while loops before it even reaches the compiler, so no simplification to `m` can take place.

Let us imagine that we use a system which allows us to implement the matrix multiplication in such a way that it can inspect its arguments to look for uses of `id`. This would cover the simple case `m*id` but we would still run into problems if we complicate the use case slightly and first assign `val m2 = id` and then compute `m*m2`. To handle programs like this, it is not sufficient to just inspect the (syntactic) arguments. We need to integrate the staging expansion with some form of forward data flow propagation, otherwise the argument is just an identifier. In general, limited forms of simplification can be added (see C++ expression templates [146]) but to be fully effective, the whole range of generic compiler optimizations

(constant propagation, dead code elimination, etc) would need to be duplicated, too.

If on the other hand all we have is a facility to add new compiler passes, then we can add an optimization pass that simplifies `m*id` to `m`, but we need another pass that expands matrix multiplications into loops. This pass needs to be implemented as a low-level IR to IR transformation that is much more complicated than a straightforward staged implementation. Phase ordering problems are also likely to arise if multiple optimizations are added independently.

The deeper problem is that we are forced to commit to a single data representation. Even if we combine staging with an extensible compiler we need to make a decision: Should we treat matrices as symbolic entities with algebraic laws, implemented as IR nodes amenable to optimization? Or should we stage them so that the compiler sees just loops and arrays without abstraction overhead?

What we really want instead is to first treat operations symbolically, subject to linear algebra optimizations and generic optimizations (combined for maximum effectiveness). Once no further simplification is possible, lower the operations to expose the representation as arrays and loops and apply another set of specific optimizations (e.g. loop fusion), again combined with the usual generic ones. Since we have changed the representation, there may be new opportunities for generic optimizations, e.g. newly exposed internal sharing between two matrix or vector ops, which should become subject to common subexpression elimination again.

# Part I

# Defining Embedded Programs

# Chapter 3

# Intro: Staging as Meta Programming

How do we define programs for embedded compilers?

First a bit of terminology: Meta-programs are programs that manipulate other programs. Object-programs are programs manipulated by other programs. Meta-languages and object-languages are the languages of meta-programs and object-programs, respectively. Program generators are meta-programs that produce object-programs as their final result [130].

In the context of program generation, multi-stage programming (MSP, *staging* for short) as established by Taha and Sheard [133] allows programmers to *stage* program expressions, moving them from the meta program to the object program and delaying their evaluation until the object program is executed. The presently executing (meta program) stage acts as a program generator that composes and possibly executes the programs of the next stage. The meta program and the object programs are uniformly expressed in a single *multi-stage* program. Thus, object programs are *embedded* programs.

For the programmer, the key benefit of multi-stage programming is linguistic reuse [80]: Multi-stage programs inherit functionality of the meta-language, such as scoping and typing rules, so if a multi-stage programs type checks, all the generated code will be syntactically valid and well-typed. The goal is to allow programmers to construct a multi-stage program from a naive implementation of the same algorithm by adding staging annotations in a small number of key places. Unfortunately, staging does not generally preserve the semantics of a program [60]. Thus, it is not so easy to achieve this goal in practice because adding staging annotations can drastically change the result of even very simple and straightforward programs. A contribution of this thesis is to systematically preserve statement execution order within a stage, following earlier work on partial evaluation in the presence of effects [137]. This preserves semantics in many more cases and takes us a step closer to our goal.

**Lightweight Modular Staging (LMS)** The following chapters present the front end part of Lightweight Modular Staging (LMS), a new multi-stage programming approach developed as part of this thesis [110, 111]. The elaboration continues in Part II, which presents the back end parts.

The classical introductory staging example is to specialize the power function for a given exponent, assuming that a program will take many different numbers to the same power.

Considering the usual implementation,

```
def power(b: Double, n: Int): Double =
  if (n == 0) 1.0 else b * power(b, n - 1)
```

we want to turn the base `b` into a *staged* expression. LMS uses types to distinguish the computational stages. In particular, we change `b`'s declared type from `Double` to `Rep[Double]`. The meaning of having type `Rep[Double]` is that `b` *represents* a computation that will yield a `Double` in the next stage. We also change the function's return type accordingly.

Now we need to regain the ability to do arithmetic on `b`, which is no longer a plain `Double`. The second idea of LMS is to package operations on staged types as components. To make its required functionality explicit, we wrap the power function itself up as a component (a trait):

```
trait Power { this: Arith =>
  def power(b: Rep[Double], n: Int): Rep[Double] =
    if (n == 0) 1.0 else b * power(b, n - 1)
}
```

LMS shares many properties of earlier staging approaches but has some important differences:

1. In most previous approaches, the meta language and the object language are the same (homogeneous staging). Both are part of a dedicated, fixed multi-stage language. But sometimes we would like the object and the meta languages to be different, as there is a lot to be gained from simpler object languages. Where meta languages should focus on abstraction, embedded languages should provide only a minimum of abstraction facilities but lots of concrete operations with efficient translations. Similar to previous "offshoring" approaches [39], LMS allows programmers to write combined multi-stage programs and profit from linguistic reuse, even if the object and meta languages are different. Since LMS is a purely library-based staging solution, the embedded languages can also be extended and composed by the multi-stage program itself, without changing the meta language implementation.

2. Linguistic reuse has previously been exploited for scoping and typing, but not for execution order. Naturally, staging allows programmers to define execution order by adding staging annotations. But in practice, programmers must explicitly and precisely define the execution order of *all* staged program fragments! As shown in Section 6.1.6, the execution order is not related to the appearance of the statements in the multi-stage program. With our approach, programmers only define the stage. The execution order within a stage is retained from the multi-stage program. As shown in Section 6.2.1, this aspect of LMS is also applicable to more traditional staging approaches based on quasiquotation, which previously relied on monads for sequencing [129].

3. Previous staging approaches used syntactic annotations to distinguish stages. LMS reduces the syntactic overhead by using only types. This makes the system more expressive because changes required to add staging are more localized. Deep reuse of type inference provides a simple form of local binding-time analysis.

**Organization** Before we dive into the details of LMS, we take a look at language virtualization (Chapter 4) and the Scala-Virtualized language extensions (Chapter 5), which are helpful to support embedded languages and in particular library based staging in Scala.

After that, we examine how linguistic reuse allows staging to simplify the development of program generators (Chapter 6). We review the state of the art (Section 6.1), identify open problems and present the contributions of LMS in detail (Section 6.2).

# Chapter 4

# Language Virtualization

The necessity of general purpose languages to serve as meta languages is widely recognized [81, 125, 118], not only in the context of program optimizations through code generation and staging but also more generally for embedding DSLs [58, 59].

We propose language virtualization as a concept to capture necessary conditions to assess the ability of a programming language to serve as meta language. In an analogy to hardware virtualization [106], where one wants to virtualize costly "big iron" server resources in a data center and run many logical machines on top of them, it is desirable to leverage the engineering effort that went into a general-purpose language to support many small embedded languages, each of which should behave more or less like a real language. The following material is taken from [19].

## 4.1 Defining Language Virtualization

**Definition.** A programming language is *virtualizable* with respect to a class of embedded languages (and by extension programs) if and only if it can provide an environment to these embedded languages that makes the embedded implementations (and by extension programs) essentially identical to corresponding stand-alone language implementations (programs) in terms of *expressiveness*, *performance* and *safety*—with only modestly more *effort* than implementing the simplest possible complete embeddings (as pure libraries).

*Expressiveness* encompasses syntax, semantics and, in the case of domain-specific languages, general ease of use for domain experts. Just as virtual hardware resources are not exactly identical to real ones, we do not require that an embedded language can exactly model the syntax of a stand-alone language but settle for a syntax that is essentially the same, i.e. modulo syntactic sugar. The same consideration applies to the other criteria as well.

*Performance* implies that programs in the embedded language must be amenable to extensive static and dynamic analysis, optimization, and code generation, just as programs in a stand-alone implementation would be. For many embedded languages, in particular those that are the focus of this paper, this rules out any purely interpretation-based solutions.

*Safety* means that the embedded implementation is not allowed to loosen guarantees about program behavior. In particular, host-language operations that are not part of the embedded language's specification must not be available to embedded programs.

Modest *effort* is the only criterion that has no counterpart in hardware virtualization. However, it serves an important purpose since an embedded language implementation that takes a DSL program as a string and feeds it into an external, specialized stand-alone compiler would trivially satisfy criteria *expressiveness*, *performance* and *safety*. Building this implementation, however, would include the effort of implementing the external compiler, which in turn would negate any benefit of the embedding. In a strict sense, one can argue that virtualizability is not a sufficient condition for a particular language being a good embedding environment because the "simplest possible" embedding might still be prohibitively expensive to realize.

The virtualization criteria are similar to Veldhuizen's criteria for universal languages [148] but add the effort criterium. There are also considerable similarities to Steele's approach of "growing a language" [125], which demands that features provided by a library should look and behave like built-in language features. Virtualization requires explicitly that library features should not only look like built-in features but also exhibit the same performance.

### 4.1.1 Virtualization and Reflection

The ability of a programming language to represent its (meta) programs as object programs is called reflection [121]. In other words, reflection enables a program to inspect and reason about itself.

Virtualization can be seen as a (static) dual of (dynamic) reflection: Where a reflective language allows programs to inspect language elements they are composed of, a virtualizable language allows programs to give language elements new meaning. In a reflective language programs can use information obtained by reflection to trigger a certain behavior. In a virtualizable language the language elements can be customized to trigger the behavior directly within programs.

In practical terms, virtualization allows programs to override and reinterpret built-in language features. In that sense, virtualized language features are similar to virtual (i.e. overridable) methods in the terminology of C++.

## 4.2 Achieving Virtualization

What does it take to make a language virtualizable in practice? Various ways of fulfilling subsets of the requirements exist, but we are unaware of any existing language that fulfills all of them. The "pure embedding" approach [59] of implementing embedded languages as pure libraries in a modern host language can likely satisfy *expressiveness*, *safety* and *effort* if the host language provides a strong static type system and syntactic malleability (e.g. custom infix operators). Achieving *performance* in addition, however, seems almost impossible without switching to a completely different approach.

**Expressiveness** We can maintain *expressiveness* by overloading all relevant host language constructs. In Scala, for example, a for-loop such as

```
for (x <- elems if x % 2 == 0) p(x)
```

is defined in terms of its expansion

```
elems.withFilter(x => x % 2 == 0)
     .foreach(x => p(x))
```

Here, `withFilter` and `foreach` are higher-order methods that need to be defined on the type of `elems`. By providing suitable implementations for these methods, a domain-specific language designer can control how loops over domain collections should be represented and executed.

To achieve full virtualization, analogous techniques need to be applied to all other relevant constructs of the host language. For instance, a conditional control construct such as

```
if (cond) something else somethingElse
```

would be defined to expand into the method call

```
__ifThenElse(cond, something, somethingElse)
```

where `__ifThenElse` is a method with two call-by-name parameters:

```
def __ifThenElse[T](cond: Boolean, thenp: => T, elsep: => T)
```

Domain languages can then control the meaning of conditionals by providing overloaded variants of this method which are specialized to domain types.

In the same vein, all other relevant constructs of the host language need to map into constructs that are extensible by domain embeddings, typically through overloading method definitions.

**Performance** As we have argued above, achieving *performance* requires the ability to apply extensive (and possibly domain-specific) optimizations and code generation to embedded programs. This implies that embedded programs must be available at least at some point using a lifted, AST-like intermediate representation. Pure embeddings, even if combined with (hypothetical) powerful partial evaluation as suggested in [59], would not be sufficient if the target architecture happens to be different from the host language target. What is needed is essentially a variant of staged metaprogramming, where the embedded "object" program can be analyzed and manipulated by a "meta" program that is part of the embedding infrastructure. However, any DSL will also contain generic parts, some of which will be host language constructs such as function definitions, conditionals or loops. These must be lifted into the AST representation as well.

This ability to selectively make constructs 'liftable' (including their compilation) such that they can be part of (compiled) DSL programs while maintaining *expressiveness*, *safety* and *effort* is an essential characteristic of virtualizable languages.

**Modest Effort** However, having to implement the lifting for each new DSL that uses a slightly different AST representation would still violate the *effort* criterion. Using an existing multi-stage language such as MetaOCaml [53, 130] would also likely violate this criterion, since the staged representation cannot be analyzed (for safety reasons we will consider shortly) and any domain-specific optimizations would require effort comparable to a stand-alone compiler. Likewise, compile-time metaprogramming approaches such as C++ templates [144] or Template Haskell [117] would not achieve the goal, since they are tied to the same target architecture as the host language and their static nature precludes dynamic optimizations (i.e. recompilation). What is needed here is a dynamic multi-stage approach with an extensible common intermediate representation (IR) architecture. In the context of Scala, we can make extensive use of traits and mixin-composition to provide building blocks of common DSL functionality (API, IR, optimizations, code generation), including making parts of Scala's semantics available as traits. This approach, which we call lightweight modular staging [110], is described below and allows us to maintain the *effort* criterion. A key element is to provide facilities to compile a limited range of Scala constructs to architectures different from the JVM, Scala's primary target.

**Safety** There are two obstacles to maintaining *safety*. The first is to embed a typed object language into a typed meta language. This could be solved using a sufficiently powerful type system that supports an equivalent of GADTs [115, 104] or dependent types [103]. The second problem is that with a plain AST-like representation, DSL programs can get access to parts of their own structure. This is unsafe in general and also potentially renders optimizations unsound. The *finally tagless* [15] or *polymorphic embedding* [55] of DSLs that forms a basis for lightweight modular staging solves both problems at once by abstracting over the actual representation used.

# Chapter 5

# Scala-Virtualized

In this chapter, we present Scala-Virtualized, our effort to improve the meta-language capabilities of Scala. Scala is a mature language that is seeing widespread adoption in industry. Scala-Virtualized is a suite of minimal extensions, based on the same codebase and undergoing the same rigorous testing as the main Scala distribution. Part of the following material is taken from [95].

Scala is used successfully for pure library-based DSLs, such as parser combinators [94], actors [54] and testing frameworks. DSLs typically leverage the following three aspects of Scala's flexibility:

- Flexible syntax. From mimicking BNF in the parser combinator library, over concise syntax for message passing, to naturally expressing specifications in the Specs [140] testing framework:

  ```
  // generate 500 different mail addresses
  mailAddresses must pass { address =>
    address must be matching(companyPattern)
  }
  ```

  Scala is an extensible language because in many cases libraries can be made to look like built-ins. However Scala does not provide real, arbitrarily extensible syntax (such as Racket [139]).

- Redefining the run-time semantics of for-comprehensions.

  ```
  for (i <- foo) yield 2 * i
  ```

  This expression is desugared to the following method call:

  ```
  foo.map(i => 2 * i)
  ```

  The class of `foo` defines the implementation and type signature of the methods (such as `map` and `flatMap`) that define the semantics of a for-comprehension. This allows DSLs to provide non-standard implementations of looping constructs.

- Customizing the type system and reifying types at run time. Domain-specific type restrictions can be implemented using implicit resolution, which provides a limited form of logic programming at the type level [31]. Phantom types and other classic tricks are also commonly used. Finally, manifests [37] put static types to work at run time.

These features are sufficient for DSLs as pure libraries. However they fall short when better performance is required, additional program properties must be verified for safety, or code is to be generated for different platforms. This requires an accessible representation of embedded programs that can be analyzed and transformed.

On the other hand, an accessible program representation alone, such as provided by a lifting mechanism that reifies expression trees, is not always sufficient. First, reified expression trees can contain arbitrary host language expressions, not just those that are also part of the embedded DSL. Second, in many cases it is desirable to freely mix lifted DSL code and non-lifted host language code.

## 5.1 Everything is a Method Call

The overarching idea of embedded languages is that user-defined abstractions should be first class in a broad sense. User-defined abstractions should have the same rights and privileges as built-in abstractions. Scala-Virtualized redefines many built-in abstractions as method calls. In this way, the corresponding method definitions may be redefined by a DSL, just like any other method. In a sense we do not make user-defined abstractions first class but built-in abstractions second class. The net effect is the same: both have equal rights.

The essential difference between Scala-Virtualized and plain vanilla Scala is that more of a Scala program is expressed in terms of method calls. Similar to the "finally tagless" [16] or polymorphic embedding [55] approach, and going back to an old idea of Reynolds [109], we represent object programs using method calls rather than data constructors. By overriding or overloading the default implementations appropriately, the embedding can be configured to generate an explicit program representation, which is typically only provided by a deep embedding using explicit data constructors.

### 5.1.1 Virtualizing Control Structures

[1] In Scala-Virtualized an expression such as `if(c) a else b` is translated into a method call `__ifThenElse(c, a, b)`. By providing its own implementation of this method, the DSL can have it generate an AST for this part of the domain program, which can thus further be analyzed and optimized by the DSL implementation. When no alternative implementation is provided, the if-then-else has the usual semantics.

This approach fits well with the overall Scala philosophy: for-comprehensions and parser combinators were implemented like this from the beginning. Unlike approaches that lift host language expression trees 1:1 using a fixed set of data types, the DSL implementor has control over which language constructs are lifted and which are not.

[1] *Credits:* Design by the author, original implementation with help from Martin Odersky.

To give an (admittedly silly) example, we could change **if** to print its condition and return the then-branch, discarding the else-branch:

```
scala > def __ifThenElse[T](cond: Boolean, thenp: => T, elsep: => T): T
                = { println("if: "+cond); thenp }
__ifThenElse: [T](cond: Boolean, thenp: => T, elsep: => T)T

scala > if(false) 1 else 2
// virtualized to: `__ifThenElse(false, 1, 2)`
if: false
res0: Int = 1
```

Besides **if**, the following control structures and built-ins (left column) are virtualized into method calls (right column):

| | |
|---|---|
| `if (c) a else b` | `__ifThenElse(c, a, b)` |
| `while(c) b` | `__whileDo(c, b)` |
| `do b while(c)` | `__doWhile(b, c)` |
| `var x = i` | `val x = __newVar(i)` |
| `x = a` | `__assign(x, a)` |
| `return a` | `__return(a)` |
| `a == b` | `__equal(a, b)` |

These methods are defined as follows in a trait `EmbeddedControls`:

```
trait EmbeddedControls {
  def __ifThenElse[T](cond: Boolean, thenp: => T, elsep: => T): T
  def __whileDo(cond: Boolean, body: Unit): Unit
  def __doWhile(body: Unit, cond: Boolean): Unit
  def __newVar[T](init: T): T
  def __assign[T](lhs: T, rhs: T): Unit
  def __return(expr: Any): Nothing
  def __equal(expr1: Any, expr2: Any): Boolean
}
```

Trait `EmbeddedControls` is mixed into `Predef`, which is implicitly imported into every compilation unit. Thus, the virtualization hooks are available in any Scala program. Programmers can either shadow them by defining a synonymous method, or override/overload them by inheriting from a DSL trait that mixes in `EmbeddedControls`.

### 5.1.2 Virtualizing Method Calls

Ordinarily, there are two ways to customize the meaning of an expression such as `x a y`, which is short for `x.a(y)`. Obviously, if we control the type of `x`, we can simply introduce the appropriate method in its class. Otherwise an implicit conversion can be used — if (and only if) `x`'s type does not provide an (appropriately typed) member `a`. While useful, this technique requires a fair bit of boilerplate code and it imposes some code size and run-time overhead. Most importantly, it cannot be used to override existing methods, such as the ubiquitous method `toString`.

Overriding existing behavior is necessary in a number of cases for embedded DSLs:

```
val buffer = new ArrayBuffer[Int]
trait DSL {
  buffer += 7
}
```

Here, class `ArrayBuffer` already has a `+=` method but we want to override the behavior within the scope of trait `DSL` to construct a staged representation. If we do not, 7 will be added to the buffer at staging time and not when running the staged program, as would be desired.

Scala-Virtualized introduces infix functions that are able to selectively and externally introduce new methods on existing types, as well as override existing ones, without any runtime overhead. The idea is simple: We redefine `x.a(y)` as `infix_a(x,y)`. If the type of `x` has any members with name `a`, we insert them as sentinels of the form **`def`** `infix_a(x,y)` into `EmbeddedControls`. If overloading resolution picks one of the sentinels, the regular invocation `x.a(y)` is chosen. Otherwise a user-defined method takes precedence.

Infix functions can greatly reduce boilerplate in many cases. Complementary to Scala's syntactic flexibility, Scala's type system also allows enforcing certain restrictions. For example, it may be desirable to restrict DSL expressions to a given grammar. Here is an example how adherence of DSL expressions to a context-free grammar ($a^n b^n$) can be enforced using phantom types and infix functions:

```
object Grammar {
  type ::[A,B] = (A,B)
  class WantAB[Stack] extends WantB[Stack]
  class WantB[Stack]
  class Done
  def start() = new WantAB[Unit]
  def infix_a[Stack](s: WantAB[Stack]) = new WantAB[Unit::Stack]
  def infix_b[Rest](s: WantB[Unit::Rest]) = new WantB[Rest]
  def infix_end(s: WantB[Unit]) = new Done
  def phrase(x: => Done): String = "parsed"
}
import Grammar._
phrase { start () a () a () b () b () end () }      // "parsed"
phrase { start () a () a () b () b () b () end () } // error
phrase { start () a () a () b () end () }           // error
```

The same behavior can be encoded in vanilla Scala using implicit conversions but in a more cumbersome way.

### 5.1.3 Virtualizing Record Types

[2] The Scala-Virtualized compiler can turn an expression **`new`** `C{`**`val`** `x_i: T_i = v_i}` into a method call `__new( ("x_i", (self_i: R) => v'_i) )`. There is no definition of `__new` in

[2] *Credits:* Design by the author and Adriaan Moors, with help from Hassan Chafi, Nada Amin, Grzegorz Kossakowski, implementation by Adriaan Moors.

`EmbeddedControls`, as its signature would be too unwieldy. Virtualization is not performed unless there exists a type constructor `Rep`, such that `C` is a subtype of `Struct[Rep]`, where the marker trait `Struct` is defined in `EmbeddedControls`:

```
trait Struct[+Rep[x]]
```

Furthermore, for all `i`,

- there must be some `T'_i` so that `T_i = Rep[T'_i]` – or, if that previous equality is not unifiable, `T_i = T'_i`
- `v'_i` results from retyping `v_i` with expected type `Rep[T'_i]`, after replacing **`this`** by a fresh variable `self_i` (with type `Rep[C{ val x_i: T'_i }]`, abbreviated as `R`)

Finally, the call `__new(("x_i", (self_i: R) => v'_i))` must type check with expected type `R`. If this is the case, the **`new`** expression is replaced by this method call. This assumes a method in scope whose definition conforms to:
`def __new[T](args: (String, Rep[T] => Rep[_])*): Rep[T]`.

#### Type-Safe Selection on Record Fields

In addition to virtualizing object creation, Scala-Virtualized provides a facility for type-safe access of record fields. When `e` refers to a representation of a record, `e.x_i` is turned into `e.selectDynamic[T_i]("x_i")` as follows. When a selection `e.x_i` does not type check according to the normal typing rules, and `e` has type `Rep[C{ val x_i: T_i }]` (for some `Rep` and where `C` and the refinement meet the criteria outlined above), `e.x_i` is turned into `e.selectDynamic[T_i]("x_i")`. Note the `T_i` type argument: by defining `selectDynamic` appropriately, the DSL can provide type safe selection on records. No type argument will be supplied when the field's type cannot be determined, i.e., it is not in the record's refinement.

### 5.1.4 Virtualizing Pattern Matching

[3] Pattern matching expressions can also be virtualized. Let us consider a simple match expression:

```
7 match { case 5 => "foo" case _ => "bar" }
```

The virtualized Scala compiler will translate this expression to:

```
__match.runOrElse(7) { x =>
  __match.guard(x == 5, "foo").orElse(__match.one("bar"))
}
```

In line with other virtualized features the translation is entirely structural. A suitable `__match` object that provides the expected methods must be available in scope. The translation target is modeled after the zero-plus monad and distinguishes between pure types, which denote the

[3] *Credits:* Design and implementation mostly by Adriaan Moors, the author helped with some details of the design.

domain of values matching is performed on, and monadic types, which are used to structure the matcher logic.

The default implementation looks like this:

```
type Pure[T]  = T
type Monad[T] = Option[T]

object __match {
  def zero: Monad[Nothing] = None
  def one[T](x: Pure[T]): Monad[T] = Some(x)
  def guard[T](cond: Pure[Boolean], then: => Pure[T]): Monad[T] =
    if(cond) Some(then) else None
  // runs the matcher on the given input
  def runOrElse[T, U](in: Pure[T])(matcher: Pure[T] => Monad[U]): Pure[U] =
    matcher(in) getOrElse (throw new MatchError(in))
  // used for isDefinedAt
  def isSuccess[T, U](x: Pure[T])(f: Pure[T] => Monad[U]): Pure[Boolean] =
    !f(x).isEmpty
}
```

Internally, the Scala-Virtualized compiler will apply all the usual optimizations if the default (non-virtualized) case is detected, such as translating match expressions to conditionals and generating jumps instead of method calls.

A virtualized pattern matching interface with `Rep` types suitable for staging can look like this:

```
val __match: Matcher

abstract class Matcher {
 def runOrElse[T, U](in: Rep[T])(matcher: Rep[T] => Maybe[U]): Rep[U]

 def zero: Maybe[Nothing]
 def one[T](x: Rep[T]): Maybe[T]
 def guard[T](cond: Rep[Boolean], then: => Rep[T]): Maybe[T]
 def isSuccess[T, U](x: Rep[T])(f: Rep[T] => Maybe[U]): Rep[Boolean]
}

abstract class Maybe[+A] {
 def flatMap[B](f: Rep[A] => Maybe[B]): Maybe[B]
 def orElse[B >: A](alternative: => Maybe[B]): Maybe[B]
}
```

## 5.2 Putting Static Information to Work

Scala's implicits [31] provide a convenient way of deriving run-time information from static types. When the last argument list of a method is marked as **`implicit`**, a call to this method need not specify its actual arguments. For each missing implicit argument, the compiler will

(statically) determine the (unique) implicit value of the correct type in order to complete the method call. The **implicit** keyword is used to mark regular value definitions as potential implicit arguments. By overriding a virtualized language feature to include certain implicit parameters we can require additional static information or predicate virtualization on some static condition.

### 5.2.1 Virtualizing Static Type Information

Certain types of implicit values are treated specially by the compiler: when no user-defined implicit value of the expected type can be found, the compiler synthesizes the value itself. In standard Scala, manifests, which provide a run-time representation of static types, are the only implicit values that are treated this way [37].

As an example of manifests, consider the following polymorphic method that requires a manifest for its type parameter `T`:

```
def m[T](x: T)(implicit m: Manifest[T]) = ...
```

When this method is called at type `String`, and assuming there is no implicit value of type `Manifest[String]` in scope, the compiler will synthesize a factory call that generates a run-time representation of the class `String`, like this:

```
reflect.Manifest.classType(classOf[String])
```

The main use of manifests in the context of embedded DSLs is to preserve information necessary for generating efficient specialized code in those cases where polymorphic types are unknown at compile time (e.g., to generate code that is specialized to arrays of a primitive type, even though the object program is constructed using generic types).

### 5.2.2 Virtualizing Static Source Information

[4] Scala-Virtualized extends the idea of `Manifest` and introduces `SourceContext` to provide run-time information about the static *source code* context. Implicit source contexts reify static source information, such as the current file and line number, which is otherwise lost after the program is compiled. The idea is for a method to declare an implicit parameter of type `SourceContext`:

```
def m[T](x: T)(implicit pos: SourceContext) = ...
```

Inside the method `m`, the source context of its invocation, i.e., the file name, line number, character offset, etc., is available as `pos`. Like manifests, source contexts are generated by the compiler on demand.

Implicit `SourceContext` object are chained to reflect the static call path. Thus they can provide source information that is impossible to recover from exception stack traces, say. Consider the following example:

[4] *Credits:* Design by the author and Philipp Haller after an initial idea by Ingo Maier, implementation by Philipp Haller.

```
def m()(implicit pos: SourceContext) = ...
def outer()(implicit outerPos: SourceContext) =
  () => m()
val fun = outer()
fun() // invoke closure
```

Here, the method `outer` returns a closure which invokes method `m`. Since `m` has an implicit `SourceContext` parameter, the compiler generates an object containing source information for the invocation of `m` inside the closure. The compiler will not only pass the `SourceContext` corresponding to the current invocation but also the `outerPos` context as the parent of the current `SourceContext`. As a result, when invoking the closure inside `m` the chain of source contexts remains available. Both inside `m` as well as inside the closure, the static source context of the closure is known. This means that even if the closure escapes its static creation site, when the closure is invoked, the source context of its creation site can be recovered. Stack traces would not be able to expose this information since it can not be recovered from the dynamic call stack.

# Chapter 6

# Staging: Deep Linguistic Reuse for Easier Program Generator Development

The idea of linguistic reuse was first introduced in the context of syntactic extensions for first-class components [80]. An embedded language should reuse the features of its host language. We distinguish mostly syntactic *shallow* reuse from *deep* reuse that includes a nontrivial translation step: We consider reuse of a linguistic feature as *deep* if the feature is translated away and the embedded language does not need to implement it. This is in contrast to shallow reuse provided directly by virtualization, for example reusing the familiar syntax `var x = 7` in DSL programs to express a creation of an object-program variable via `__newVar(7)`. In this case, the feature of variables is reused but not translated away. The object language still has to accommodate mutable variables.

In the context of staging, deep linguistic reuse plays a key role in reducing the development effort for program generators. We first review the state of the art in multi-stage programming with an eye on deep reuse (Section 6.1) and then present our contributions in Section 6.2.

## 6.1 State of the Art

Previous staging approaches either work directly with strings that represent concrete program syntax or make use of quasiquoting to compose abstract syntax trees. We examine both approaches in turn, with an eye on how linguistic reuse improves productivity and safety for the multi-stage programmer.

### 6.1.1 Program Generation with Strings

As a simple example, let us turn the power function:

```
def power(b: Double, n: Int): Double =
  if (n == 0) 1.0 else b * power(b, n - 1)
```

into a code generator:

```
def power(b: String, n: Int): String =
  if (n == 0) "1.0" else "(" + b + " * " + power(b, n - 1) + ")"
```

As result of an invocation we obtain:

```
power("x",4) // "(x * (x * (x * (x * 1.0)))"
```

However there is a problem: We can produce arbitrary strings that might not be valid code. It is very easy to make subtle mistakes:

```
def power(b: String, n: Int): String =
  if (n == 0) "1.0" else "b * " + power(b, n - 1) + ")"
```

We have accidentally omitted a parenthesis, so the result is not syntactically well formed code. Furthermore, the literal identifier `b` is part of the output:

```
power("x",4) // "b * b * b * b * 1.0)))"
```

This code will not compile and even if we fix the syntax, the code is no longer well scoped. The free identifier *b* can lead to variable capture when the code is spliced in somewhere else.

We have seen two problems, syntax correctness and scope correctness. Two other problems are type correctness and value correctness. If we cannot guarantee to generate valid programs, we can much less guarantee that programs are well-typed or compute correct results.

### 6.1.2 Program Generation with Quasi-Quotes

Strings model concrete syntax, but we can also use abstract syntax. This idea is inspired by Lisp's "code as data" model. We start with a slightly more convenient string notation, denoted by `s"..."` quotes:

```
def power(b: String, n: Int): String =
  if (n == 0) s"1.0" else s"($b * ${ power(b, n - 1) })"
```

The notation `${ ... }` denotes a hole in the string, to be filled by the string result of evaluating the enclosed expression.

The same idea applies to abstract syntax. Let `[[ ... ]]` denote the AST of the enclosed expression, and let `Tree` be the type of AST nodes. Holes will require an expression of type `Tree`:

```
def power(b: Tree, n: Int): Tree =
  if (n == 0) [[1.0]] else [[$b * ${ power(b, n - 1) }]]"
```

Now we have a program generator that assembles AST nodes.

### 6.1.3 Syntactic Correctness through Deep Reuse of Syntax

The multi-stage language compiler parses the whole program and builds ASTs for all expressions, quoted or not, at once. Thus we obtain syntax correctness. However the multi-stage

language compiler must know about the syntax of the object language. This is trivial if meta-language and object language are the same. Otherwise it is slightly more difficult [86].

The `Tree` type can be left abstract. Some implementations hide the exact data structures to guarantee safety of optimizations on object code. Silently modifying trees with rewrites that maintain semantic but not structural equality (e.g. beta reduction) can change the behavior of programs that inspect the tree structure [131]. In general, optimizations should not change the result of a program.

### 6.1.4 Scope Correctness through Deep Reuse of Scope

The multi-stage compiler can bind identifiers at the definition site of the quote. This avoids variable capture and ensures scope correctness (hygiene).

Another possible issue is *scope extrusion*. This happens when a variable bound in quoted code escapes through a hole:

```
var x: Tree;
[[ val y = 7; ${ x = y }]]
```

Scope extrusion can be prevented by appropriate type systems [155, 70], which are beyond the scope of this thesis. Scope extrusion is a real problem for code generators that imperatively manage staged program fragments. For generators expressed in a functional style it it far less of an issue, regardless of whether the object program uses effects or not.

### 6.1.5 Type Correctness through Deep Reuse of Types

With syntax and scoping out of the way, we turn our attention to type correctness. Fortunately, type correctness falls out naturally if parametric types are available. We just replace type `Tree` with `Tree[T]`:

```
def power(b: Tree[Double], n: Int): Tree[Double] =
```

Now the type system ensures that `power` is only applied to AST nodes that compute `Double` values in the next stage.

Note that the use of parametric types alone does not prevent scope extrusion, which can also be seen as a type error in the sense of a well-typed multi-stage program "going wrong" [134, 132]. Thus we do not obtain a guarantee that *all* generated programs type check, but the slightly weaker assurance that all generated programs that are well-formed are also type correct.

### 6.1.6 Value Correctness is an Open Problem

The remaining big problem is what we (somewhat vaguely) call *value correctness* or more generally preservation of program semantics: How can we be reasonably certain that a program computes the same result after adding staging annotations? We cannot expect a strong guarantee in all cases for reasons of nontermination but what is troubling is that there are many practical cases where staging annotations change a program's behavior quite drastically. This fact is well documented in the literature [40, 25, 14, 129, 60].

The problem manifests itself both with strings and with trees. The root cause is that both approaches are based on syntactic expansion, irrespective of semantics such as order of execution.

Using the regular, unstaged power implementation:

```
def power(b: Double, n: Int): Double = ...

val x = computeA()       // computeA executed here
power(computeB() + x, 4) // computeB executed before calling power (cbv)
```

Both compute functions will be executed once, in order. Afterwards, power will be applied to the result.

Let us compare this with the staged implementation:

```
def power(b: Tree[Double], n: Int): Tree[Double] = ...

val x = [[computeA()]]
power([[computeB() + $x]], 4)
```

Result:

```
((computeB() + computeA()) *
 ((computeB() + computeA()) *
  ((computeB() + computeA()) *
   ((computeB() + computeA()) * 1.0))))"
```

In this case, the computation has been duplicated $n$ times and the order of the function calls has been reversed. Effectively we ignore all bindings and follow a call-by-name policy even though power declares its arguments as call-by-value. If either of the compute functions depends on side effects the staged function computation will produce a very different result. Imagine for example:

```
def computeA() = readNextInputValue()
```

We clearly want to read only one value, not four.

Even if both functions are pure, it will be much more expensive to compute the result. If we applied staging to obtaining better performance we have not achieved our goal.

As another example, let us switch to a better algorithm:

```
def power(b: Tree[Double], n: Int): Tree[Double] =
  if (n == 0) [[ 1.0 ]]
  else if ((n&1) == 0) { val y = power(b, n/2); [[$y * $y]] }
  else [[$b * ${ power(b, n - 1) }]]
```

Result:

```
power([[x]]) // (((x*1.0)*(x*1.0))*((x*1.0)*(x*1.0)))
```

Staging has turned the more efficient algorithm into a less efficient one. This effect of staging undoing binding and memoization is widely known [129, 40].

#### Let Insertion as a Remedy

One way of fixing the order of staged expressions is to insert let-bindings in strategic places. This is frequently done by separate front ends. Staging effectively becomes an "assembly language" for code generation. The front end can assemble pieces of generated code using explicit side effects, or the code generators are written in monadic style or continuation passing style (CPS), in which case the monadic bind operation will insert let-bindings to maintain the desired evaluation order [129]. Effectful code generators are much more likely to cause scope extrusion. Explicit monadic style or CPS complicate code generators a lot. This dilemma is described as an "agonizing trade-off", due to which one "cannot achieve clarity, safety, and efficiency at the same time" [70]. Only very recently have type-systems been devised to handle both staging and effects [69, 70, 155]. They are not excessively restrictive but not without restrictions either. Mint [155], a multi-stage extension of Java, restricts non-local operations within escapes to final classes which excludes much of the standard Java library. Languages that support both staging and first class delimited continuations can mitigate this overhead but front ends that encapsulate the staging primitives are still needed [70].

In the partial evaluation community, specialization of effectful programs has been achieved by inserting let-bindings eagerly for each effectful statement [137, 82], achieving on-the-fly conversion to administrative normal form (ANF, [46]). As we will show below, a simplified variant of this approach naturally extends to staging with and without quasiquotes.

## 6.2 Contributions

We first show how to maintain value correctness through deep reuse of evaluation order. The key idea is similar to that employed in partial evaluation [137, 82] and applies to both quasiquoting and LMS. Our presentation differs from the partial evaluation literature in that it is independent of any partial evaluation mechanics such as CPS conversion and expressed in a simple, purely operational way. Continuing with quasiquoting, we show how we can remove syntactic overhead and arrive at a more restricted object language by providing a typed API over staged `Rep[T]` values that hides the internal implementation. At this point, quasiquoting becomes an implementation detail that is no longer strictly needed because the higher level object language interface has taken over most of the staging guarantees. Staging can be implemented as a library, without specific language support. Linguistic reuse is enabled by lifting operations from type `T` to `Rep[T]`. The object language can be divided into reusable components. Since there is only a single shared `Rep[T]` type, no layerings or translations between components are necessary. Deep reuse of type inference enables a form of semi-automatic local BTA since method overloading will select either staged or unstaged operations depending on the types. In many cases, methods can be staged by just changing their parameter types.

### 6.2.1 Value Correctness through Deep Reuse of Evaluation Order

The key idea is to treat quoted fragments as context-sensitive statements, not context-free expressions. This means that we will need to explicitly *perform* a statement. We continue the description with strings as the representation type since it is the most basic. Performing a statement will register the side effects of this statement in the current context. The counterpart to *perform* is *accumulate*, which defines such a context and returns a program fragment that captures all the effects within the context. To make sure that all code fragments are treated in this way we introduce the following typings:

```
type Code
def perform(stm: String): Code
def accumulate(res: => Code): String
```

Note the by-name argument of accumulate. The `Code` type and the method implementations can remain abstract for the moment.

We can put perform and accumulate to use in the power example as follows:

```
def power(b: Code, n: Int): Code =
  if (n == 0) perform("1.0") else
  perform("(" + accumulate { b } + " * " + accumulate { power(b, n - 1) } + ")")
```

We define perform and accumulate in the following way to perform automatic eager let insertion. The private constructor `code` builds a `Code` object from a string:

```
accumulate { E[ perform("str") ] }
    ⟶ "{ val fresh = str; " + accumulate { E[ code("fresh") ] } + "}"
accumulate { code("str") }
    ⟶ "str"
```

Where `E` is an accumulate-free evaluation context and `fresh` a fresh identifier. These rules can be implemented using one piece of mutable state in a straightforward way.

We are reusing the execution order of the meta language: In the meta language we execute perform whenever we encounter an object program expression. If we force the object program to replay the order of the perform calls by inserting a let binding for each of them, we are sure to execute the performed statements in the right order. Whenever we have a hole to fill in an object program fragment, we use accumulate to gather all statements performed while computing the fragment to splice into the hole.

Perform and accumulate form a reflect/reify pair that translates between a syntactic and a semantic layer. Alternatively, perform could be called reflectEffects, accumulate reifyEffects. This hints at the view that we are embedding perform and accumulate in the (invisible) computation monad of the meta language using Filinski's notion of monadic reflection [44, 45]. Accumulate is a left inverse of perform with respect to extensional equality (≡) of the generated code:

```
accumulate { perform("a") } ⟶* "{ val fresh = a; fresh }"    ≡    "a"
```

If structural equality is desired, a simple special case can be added to the above definition to directly translate `accumulate(perform("a"))` to `"a"`. Within a suitable context, perform is

also a left inverse of accumulate: Performing a set of accumulated statements together is the same as just performing the statements individually.

Clearly, using perform and accumulate manually is tedious. However we can incorporate them entirely inside the quasi quotation / interpolation syntax:

```
s" foo ${ bar } baz " ⟶ " foo " + bar + " baz " // regular interpolation
q" foo ${ bar } baz " ⟶ perform(" foo " + accumulate { bar } + " baz ")
```

In essence, we identify quotation with perform and holes with accumulate.

We get back to a power implementation using quasiquotes. This time we use type Code, although we are still working with concrete syntax:

```
def power(b: Code, n: Int): Code =
  if (n == 0) q"1.0" else  q"($b * ${ power(b, n - 1) } )"
```

The same mechanism can be used to implement order preserving versions of (type-safe) abstract syntax quotation [[...]]. The signatures will change from strings to trees:

```
type Code[T]
def perform(stm: Tree[T]): Code[T]
def accumulate(res: => Code[T]): Tree[T]
```

We put the modified quasiquotes to test by invoking power on the example from the previous Section 6.1.6:

```
def power(b: Code[T], n: Int): Code[T] = ...

val x = [[computeA()]]
power([[computeB() + $x]], 4)
```

We obtain as intermediate result before invoking power (dropping unnecessary braces and replacing semicolons with newlines):

```
val x1 = computeA()
val x3 = { val x2 = computeB() + x1; x2 }
power(x3,4)
```

And as the final result:

```
val x1 = computeA()
val x3 = { val x2 = computeB() + x1; x2 }
val x4 = 1.0
val x5 = x3 * x4
val x6 = x3 * x5
val x7 = x3 * x6
val x8 = x3 * x7
x8
```

It is easy to see that this is the correct sequencing of statements. No computation is duplicated. Likewise, if we use the improved algorithm, we actually get better performance.

We have removed the need for monadic or side-effecting front-ends (in this case, in other cases they may still be needed but never to perform let insertion). Since we have extended

the core staging primitives with a controlled form of side effect, we have removed the need for uncontrolled side effects in the generator. This makes otherwise common errors such as scope extrusion much less likely.

### 6.2.2 Removing Syntactic Overhead

We have seen how we can improve staging based on quasiquotes or direct string generation. Now we turn to other approaches of delineating embedded object programs. Our aim is embedding domain specific compilers. We want object languages tailored to specific applications, with custom compiler components. The "one size fits all" approach of having the same meta and object language is not ideal for this purpose. In our case, we would have to inspect Scala ASTs and reject or possibly interpret constructs that have no correspondence in the object language (type, class or method definitions, etc).

The staged power implementations with quasi quotes (Sections 6.1.2,6.1.6) look OK but they do contain a fair bit of syntactic noise. Also, we might want stronger guarantees about the staged code, for example that it does not use a particular language feature, which we know is detrimental to performance. What is more, we might want to generate code in a different language (JavaScript, CUDA, SQL).

We already hide the internal code representation from client programs. There are good reasons to also hide the full power of arbitrary program composition / quasi quoting from client programs.

Programs, such as power, use quasiquotes for two purposes: lifting primitives and operations:

```
def power(b: Code[Double], n: Int): Code[Double] =
  if (n == 0) q"1.0" else  q"($b * ${ power(b, n - 1) } )"
```

We already identify object code via `Code[T]` types. Instead of quasiquotes we can employ other ways of lifting the necessary operations on type `T` to type `Code[T]`:

```
implicit def liftDouble(x: Double): Code[Double] = q"x"
def infix_*(x: Code[Double], y: Code[Double]): Code[Double] = q"$x * $y"
```

Now power can be implemented like this:

```
def power(b: Code[Double], n: Int): Code[Double] =
  if (n == 0) liftDouble(1.0) else infix_*(b, power(b, n - 1))
```

But we can simplify further. In fact, the Scala compiler will do most of the work for us and we can write just this:

```
def power(b: Code[Double], n: Int): Code[Double] =
  if (n == 0) 1.0 else b * power(b, n - 1)
```

Apart from the `Code[_]` types, we have re-engineered exactly the regular, unstaged power function! All other traces of staging annotations are gone.

We are relying on Scala's support for implicit conversions (views) and Scala-Virtualized support for infix methods. Other expressive languages provide similar features.

### 6.2.3 Staging as a Library and Modular Definition of Object Languages

With the object language definition given by method signatures we can implement staging as a library, without dedicated language support and with roughly the same guarantees as a multi-stage language with quasi quotation. Furthermore, we can easily generate code in another target language, for example emit JavaScript from staged Scala expressions. Given that the multi-stage program is in control of defining the object language we can model additional guarantees about the absence of certain operations from staged code, simply by not including these operations in the object language interface.

The core idea is to delegate correctness issues to the implementations of the lifted operations, i.e. the implementation of the object language interface. Client code can access staging only through the object language API, so if the implementation is correct, the interface ensures correctness of the client code.

We can use any representation we like for staged expressions. For the sake of simplicity we will stick to strings. Where we have used type `Code[T]` above, we will use `Rep[T]` from now on because we want to allude to thinking more about the *representation* of a `T` value in the next stage and less about composing code fragments.

Where quasiquoting allowed the full language to be staged, we now have to explicitly "white-list" all operations we want to make available. Clearly there is a tradeoff, as explicit white-listing of operations can be tedious. However we can remedy the white-listing effort to a large extent by providing libraries of reusable components that contain sets of lifted operations from which different flavors of object languages can be assembled. It is also possible to lift whole traits or classes using reflection [79].

We can define a simple object language `MyStagedLanguage` as follows, using **`private`** access qualifiers to ensure that the staging primitives perform and accumulate are inaccessible to client code outside of package `internal`:

```
package internal
trait Base extends EmbeddedControls {
  type Rep[T]
  private[internal] def perform[T](stm: String): Rep[T]
  private[internal] def accumulate[T](res: => Rep[T]): String
}
trait LiftPrimitives extends Base {
  implicit def liftDouble(x: Double): Rep[Double] = perform(x.toString)
}
trait Arith extends Base {
  def infix_*(x: Rep[Double], y: Rep[Double]): Rep[Double] = perform(x+"*"+y)
}
trait IfThenElse extends Base {
  def __ifThenElse[T](c: Rep[Boolean], a: =>Rep[T], a: =>Rep[T]): Rep[T] =
    perform("if (" + c + ") " + accumulate(a) + " else " + accumulate(b))
}
trait MyStagedLanguage extends LiftPrimitives with Arith with IfThenElse
```

Note that we invoke accumulate only for by-name parameters. All others are already object

code values, so evaluating them is a no-op and cannot have side effects. In doing so we silently assume a sensible `toString` operation on `Rep[T]`. If we do not want to make this assumption then we need accumulate calls everywhere a `Rep[T]` value is converted to a string representation.

Client code just needs access to an object of type `MyStagedLanguage` to call methods on it. Common ways to achieve this include path-dependent types and imports:

```
val p: MyStagedLanguage = ...
import p._
def power(b: Rep[Double], n: Int): Rep[Double] = ...
```

In which case the desugared method signature is:

```
def power(b: p.Rep[Double], n: Int): p.Rep[Double] = ...
```

Or by structuring the client code as traits itself:

```
trait Power { this: MyStagedLanguage =>
  def power(b: Rep[Double], n: Int): Rep[Double] = ...
}
```

In the following we briefly revisit the various static guarantees and show how they are fulfilled in LMS.

#### Syntax correctness through Embedding as Methods

Generating syntactically well formed programs is delegated to methods implementing the object language interface. Client code never assembles pieces of code directly. If clients only use the API methods, and their implementations produce syntax correct code, overall syntax correctness follows.

#### Scope Correctness through Deep Reuse Of Val Bindings

The staging primitives perform eager let insertion and perform will assign a fresh identifier to each and every subexpression encountered, essentially producing on object program in administrative normal form (ANF). This removes the need for explicit val bindings in object code. Instead, programmers can just use val bindings in the meta program. This is an example of deep linguistic reuse, as the "feature" of val bindings is translated away.

As for scope correctness, we have not encountered any binders in object code so far. Below in Section 6.2.4 we will introduce staged functions using higher order abstract syntax (HOAS) [105]:

```
def lambda[A,B](f: Rep[A] => Rep[B]): Rep[A=>B]
lambda { (x:Rep[Int]) => ... }  // a staged function object
```

The essence of HOAS is to reuse meta language bindings to implement object language bindings. Unless subverted by explicit scope extrusion, the reuse of meta language bindings ensures scope correctness of object programs.

#### Type Correctness through Typed Embedding (Deep Reuse of Types)

The object language API exposes only typed methods. If the implementations of these methods produce type correct code, then overall type correctness follows.

#### Value Correctness through Deep Reuse of Evaluation Order

The perform and accumulate abstraction has been described at length in Section 6.2.1.

### 6.2.4 Functions and Recursion

Many features can be added to the object language in a way that is analogous to what we have seen above but some require a bit more thought. In this section we will take a closer look at staged functions. Basic support for staged function definitions and function applications can be defined in terms of a simple higher-order abstract syntax (HOAS) [105] representation, similar to those of Carette et al. [16] and Hofer et al. [55].

The idea is to provide a `lambda` operation that transforms present-stage functions over staged values (type `Rep[A] => Rep[B]`) to staged function values (type `Rep[A=>B]`).

```
trait Functions extends Base {
  def lambda[A,B](f: Rep[A] => Rep[B]): Rep[A=>B]
  def infix_apply[A,B](f: Rep[A=>B], x: Rep[A]): Rep[B]
}
```

To give an example, the staged recursive factorial function will look like this:

```
def fac: Rep[Int => Int] = lambda { n =>
  if (n == 0) 1
  else n * fac(n - 1)
}
```

As opposed to the earlier power example, an invocation `fac(m)` will not inline the definition of `fac` but result in an actual function call in the generated code.

However the HOAS representation has the disadvantage of being opaque: there is no immediate way to "look into" a Scala function object. If we want to treat functions in the same way as other program constructs, we need a way to transform the HOAS encoding into our string representation. We can implement `lambda(f)` to call

```
accumulate { f(fresh[A]) }
```

which will unfold the function definition into a block that represents the entire computation defined by the function (assuming that `fresh[A]` creates a fresh symbol of type `A`). But eagerly expanding function definitions is problematic. For recursive functions, the result would be infinite, i.e. the computation will not terminate. What we would like to do instead is to detect recursion and generate a finite representation that makes the recursive call explicit. However this is difficult because recursion might be very indirect:

```
def foo(x: Rep[Int]) = {
  val f = (x: Rep[Int]) => foo(x + 1)
```

```
  val g = lambda(f)
  g(x)
}
```

Each incarnation of `foo` creates a new function `f`; unfolding will thus create unboundedly many different function objects.

To detect cycles, we have to *compare* those functions. This, of course, is undecidable in the general case of taking equality to be defined extensionally, i.e. saying that two functions are equal if they map equal inputs to equal outputs. The standard reference equality, by contrast, is too weak for our purpose:

```
def adder(x:Int) = (y: Int) => x + y
adder(3) == adder(3)
↪ false
```

However, we can approximate extensional equality by intensional (i.e. structural) equality, which is sufficient in most cases because recursion will cycle through a well defined code path in the program text. Testing intensional equality amounts to checking if two functions are defined at the same syntactic location in the source program and whether all data referenced by their free variables is equal. Fortunately, the implementation of first-class functions as closure objects offers (at least in principle) access to a "defunctionalized" data type representation on which equality can easily be checked. A bit of care must be taken though, because the structure can be cyclic. On the JVM there is a particularly neat trick. We can serialize the function objects into a byte array and compare the serialized representations:

```
serialize(adder(3)) == serialize(adder(3))
↪ true
```

With this method of testing equality, we can implement *controlled* unfolding. Unfolding functions only once at the definition site and associating a fresh symbol with the function being unfolded allows us to construct a block that contains a recursive call to the symbol we created. Thus, we can create the expected representation for the factorial function above.

### 6.2.5 Semi-Automatic BTA through Deep Reuse of Type Inference

Given a method or function implementation:

```
def power(b: _, n: Int) =
  if (n == 0) 1.0 else b * power(b, n - 1)
```

Scala's type inference can determine whether the operations and the result will be staged or not. We just have to provide the binding time for parameter `b`. Note that staging `n` would require explicit use of `lambda` because there is no static criterion to stop the recursion.

In some cases we need to be conservative, for example for mutable objects:

```
var i = 0
if (c)    // c: Rep[Boolean]
  i += 1
```

The variable `i` must be lifted because writes depend on dynamic control flow. We can accomplish this by implementing the virtualized var constructor to always lift variable declarations, even if the initial right-hand side is a static value. Packaged up in a trait, it can be selectively imported:

```
trait MyProg { this: LiftVariables =>
  ... // all variables are lifted in this scope
}
```

### 6.2.6 Generating and Loading Executable Code

Code generation in LMS is an explicit operation. For the common case where generated code is to be loaded immediately into the running program, trait `Compile` provides a suitable interface in form of the abstract method `compile`:

```
trait Compile extends Base {
  def compile[A,B](f: Rep[A] => Rep[B]): A=>B
}
```

The contract of `compile` is to "unstage" a function from staged to staged values into a function operating on present-stage values that can be used just like any other function object in the running program. Of course this only works for functions that do not reference externally bound `Rep[T]` values, otherwise the generate code will not compile due to free identifiers. The given encoding into Scala's type system does not prevent this kind of error.

For generating Scala code, an implementation of the compilation interface is provided by trait `CompileScala`:

```
trait CompileScala extends Compile {
  def compile[A,B](f: Rep[A] => Rep[B]) = {
    val x = fresh[A]
    val y = accumulate { f(x) }
    // emit header
    emitBlock(y)
    // emit footer
    // invoke compiler
    // load generated class file
    // instantiate object of that class
  }
}
```

The overall compilation logic of `CompileScala` is relatively simple: emit a class and `apply`-method declaration header, emit instructions for each definition node according to the schedule, close the source file, invoke the Scala compiler, load the generated class file and return a newly instantiated object of that class.

# Part II

# Compiling Embedded Programs

# Chapter 7

# Intro: Not your Grandfather's Compiler

How do embedded compilers compile their programs?

The purely string based representation of staged programs from Part I does not allow analysis or transformation of embedded programs. Since LMS is not inherently tied to a particular program representation it is very easy to pick one that is better suited for optimization. As a first cut, we switch to an intermediate representation (IR) based on expression trees, adding a level of indirection between construction of object programs and code generation (Chapter 8). On this tree IR we can define traversal and transformation passes and build a straightforward embedded compiler. We can add new IR node types and new transformation passes that implement domain specific optimizations. In particular we can use multiple passes of staging: While traversing (effectively, interpreting) one IR we can execute staging commands to build another staged program, possibly in a different, lower-level object language.

However the extremely high degree of extensibility poses serious challenges. In particular, the interplay of optimizations implemented as many separate phases does not yield good results due to the phase ordering problem: It is unclear in which order and how often to execute these phases, and since each optimization pass has to make pessimistic assumptions about the outcome of all other passes the global result is often suboptimal compared to a dedicated, combined optimization phase [149, 23]. There are also implementation challenges as each optimization needs to be designed to treat unknown IR nodes in a sensible way.

Other challenges are due to the fact that embedded compilers are supposed to be used like libraries. Extending an embedded compiler should be easy, and as much of the work as possible should be delegated to a library of compiler components. Newly defined high-level IR nodes should profit from generic optimizations automatically.

To remedy this situation, we switch to a graph-based "sea of nodes" representation (Chapter 9). This representation links definitions and uses, and it also reflects the program block structure via nesting edges. We consider purely functional programs first. A number of non-trivial optimizations become considerably simpler. Common subexpression elimination (CSE) and dead code elimination (DCE) are particularly easy. Both are completely generic and support an open domain of IR node types. Optimizations that can be expressed as context-free rewrites are also easy to add in a modular fashion. A scheduling and code motion algorithm transforms graphs back into trees, moving computations to places where they are less often

executed, e.g. out of loops or functions. Both graph-based and tree-based transformations are useful: graph-based transformations are usually simpler and more efficient whereas tree-based transformations, implemented as multiple staging passes, can be more powerful and employ arbitrary context-sensitive information.

To support effectful programs, we make effects explicit in the dependency graph (similar to SSA form). We can support simple effect domains (pure vs effectful) and more fine grained ones, such as tracking modifications per allocation site. The latter one relies on alias and points-to analysis.

We turn to advanced optimizations in Chapter 10. For combining analyses and optimizations, it is crucial to maintain optimistic assumptions for all analyses. The key challenge is that one analysis has to anticipate the effects of the other transformations. The solution is speculative rewriting [85]: transform a program fragment in the presence of partial and possibly unsound analysis results and re-run the analyses on the transformed code until a fixpoint is reached. This way, different analyses can communicate through the transformed code and need not anticipate their results in other ways. Using speculative rewriting, we compose many optimizations into more powerful combined passes. Often, a single forward simplification pass that can be used to clean up after non-optimizing transformations is sufficient.

However not all rewrites can fruitfully be combined into a single phase. For example, high-level representations of linear algebra operations may give rise to rewrite rules like $IM \rightarrow M$ where $I$ is the identity matrix. At the same time, there may be rules that define how a matrix multiplication can be implemented in terms of arrays and while loops, or a call to an external library (BLAS). To be effective, all the high-level simplifications need to be applied exhaustively before any of the lowering transformations are applied. But lowering transformations may create new opportunities for high-level rules, too. Our solution here is delayed rewriting: programmers can specify that a certain rewrite should not be applied right now, but have it registered to be executed at the next iteration of a particular phase. Delayed rewriting thus provides a way of grouping and prioritizing modularly defined transformations.

On top of this infrastructure, we build a number of advanced optimizations. A general pattern is split and merge: We split operations and data structures in order to expose their components to rewrites and dead-code elimination and then merge the remaining parts back together. This struct transformation also allows for more general data structure conversions, including array-of-struct to struct-of-array representation conversion. Furthermore we present a novel loop fusion algorithm, a powerful transformation that removes intermediate data structures.

Evaluation and examples follow in Part III.

# Chapter 8

# Intermediate Representation: Trees

With the aim of generating code, we could represent staged expressions directly as strings, as done in Part I. But for optimization purposes we would rather have a structured intermediate representation that we can analyze in various ways. Fortunately, LMS makes it very easy to use a different internal program representation.

## 8.1 Trees Instead of Strings

Our starting point is an object language *interface* derived from Part I:

```
trait Base {
  type Rep[T]
}
trait Arith extends Base {
  def infix_+(x: Rep[Double], y: Rep[Double]): Rep[Double]
  def infix_*(x: Rep[Double], y: Rep[Double]): Rep[Double]
  ...
}
trait IfThenElse extends Base  {
  def __ifThenElse[T](c: Rep[Boolean], a: =>Rep[T], b: =>Rep[T]): Rep[T]
}
```

The goal will be to build a corresponding *implementation* hierarchy that supports optimizing compilation.

Splitting interface and implementation has many advantages, most importantly a clear separation between the user program world and the compiler implementation world. For the sake of completeness, let us briefly recast the string representation from Part I in this model:

```
trait BaseStr extends Base {
  type Rep[T] = String
}
trait ArithStr extends BaseStr with Arith {
  def infix_+(x: Rep[Double], y: Rep[Double]) = perform(x + " + " + y)
  def infix_*(x: Rep[Double], y: Rep[Double]) = perform(x + " * " + y)
  ...
```

```
}
trait IfThenElseStr extends BaseStr with IfThenElse  {
  def __ifThenElse[T](c: Rep[Boolean], a: =>Rep[T], b: =>Rep[T]) =
    perform("if (" + c + ") " + accumulate(a) + " else " + accumulate(b))
}
```

In this chapter, we will use an IR that is based on expression trees, closely resembling the abstract syntax tree (AST) of a staged program. This representation enables separate analysis, optimization and code generation passes. We will use the following types:

```
type Exp[T]     // atomic:     Sym, Const
type Def[T]     // composite:  Exp + Exp, Exp * Exp, ...
type Stm[T]     // statement:  val x = Def
type Block[T]   // blocks:     { Stm; ...; Stm; Exp }
```

They are defined as follows in a separate trait:

```
trait Expressions {
  // expressions (atomic)
  abstract class Exp[T]
  case class Const[T](x: T) extends Exp[T]
  case class Sym[T](n: Int) extends Exp[T]
  def fresh[T]: Sym[T]

  // definitions (composite, subclasses provided by other traits)
  abstract class Def[T]

  // statements
  case class Stm[T](sym: Sym[T], rhs: Def[T])

  // blocks
  case class Block[T](stms: Stm[_], res: Exp[T])

  // perform and accumulate
  def reflectStm[T](d: Stm[T]): Exp[T]
  def reifyBlock[T](b: =>Exp[T]): Block[T]

  // bind definitions to symbols automatically
  // by creating a statement
  implicit def toAtom[T](d: Def[T]): Exp[T] =
    reflectStm(Stm(fresh[T], d))
}
```

This trait `Expressions` will be mixed in at the root of the object language implementation hierarchy. The guiding principle is that each definition has an associated symbol and refers to other definitions only via their symbols. This means that every composite value will be named, similar to administrative normal form (ANF). Methods `reflectStm` and `reifyBlock` take over the responsibility of `perform` and `accumulate`.

### 8.1.1 Modularity: Adding IR Node Types

We observe that there are no concrete definition classes provided by trait `Expressions`. Providing meaningful data types is the responsibility of other traits that implement the interfaces defined previously (`Base` and its descendents).

Trait `BaseExp` forms the root of the implementation hierarchy and installs atomic expressions as the representation of staged values by defining `Rep[T] = Exp[T]`:

```
trait BaseExp extends Base with Expressions {
  type Rep[T] = Exp[T]
}
```

For each interface trait, there is one corresponding core implementation trait. Shown below, we have traits `ArithExp` and `IfThenElseExp` as the running example. Both traits define one definition class for each operation defined by `Arith` and `IfThenElse`, respectively, and implement the corresponding interface methods to create instances of those classes.

```
trait ArithExp extends BaseExp with Arith {
  case class Plus(x: Exp[Double], y: Exp[Double]) extends Def[Double]
  case class Times(x: Exp[Double], y: Exp[Double]) extends Def[Double]
  def infix_+(x: Rep[Double], y: Rep[Double]) = Plus(x, y)
  def infix_*(x: Rep[Double], y: Rep[Double]) = Times(x, y)
  ...
}
trait IfThenElseExp extends BaseExp with IfThenElse {
  case class IfThenElse(c: Exp[Boolean], a: Block[T], b: Block[T]) extends Def[T]
  def __ifThenElse[T](c: Rep[Boolean], a: =>Rep[T], b: =>Rep[T]): Rep[T] =
    IfThenElse(c, reifyBlock(a), reifyBlock(b))
}
```

The framework ensures that code that contains staging operations will always be executed within the dynamic scope of at least one invocation of `reifyBlock`, which returns a block object and takes as call-by-name argument the present-stage expression that will compute the staged block result. Block objects can be part of definitions, e.g. for loops or conditionals.

Since all operations in interface traits such as `Arith` return `Rep` types, equating `Rep[T]` and `Exp[T]` in trait `BaseExp` means that conversion to symbols will take place already within those methods. This fact is important because it establishes our correspondence between the evaluation order of the program generator and the evaluation order of the generated program: at the point where the generator calls `toAtom`, the composite definition is turned into an atomic value via `reflectStm`, i.e. its evaluation will be recorded now and played back later in the same relative order with respect to others within the closest `reifyBlock` invocation.

## 8.2 Enabling Analysis and Transformation

Given our IR representation it is easy to add traversals and transformations.

### 8.2.1 Modularity: Adding Traversal Passes

All that is needed to define a generic in-order traversal is a way to access all blocks immediately contained in a definition:

```
def blocks(x: Any): List[Block[Any]]
```

For example, applying `blocks` to an `IfThenElse` node will return the then and else blocks. Since definitions are case classes, this method is easy to implement by using the `Product` interface that all case classes implement.

The basic structural in-order traversal is then defined like this:

```
trait ForwardTraversal {
  val IR: Expressions
  import IR._
  def traverseBlock[T](b: Block[T]): Unit = b.stms.foreach(traverseStm)
  def traverseStm[T](s: Stm[T]): Unit = blocks(s).foreach(traverseBlock)
}
```

Custom traversals can be implemented in a modular way by extending the `ForwardTraversal` trait:

```
trait MyTraversalBase extends ForwardTraversal {
  val IR: BaseExp
  import IR._
  override def traverseStm[T](s: Stm[T]) = s match {
    // custom base case or delegate to super
    case _ => super.traverseStm(s)
  }
}
trait MyTraversalArith extends MyTraversalBase {
  val IR: ArithExp
  import IR._
  override def traverseStm[T](s: Stm[T]) = s match {
    case Plus(x,y) => ... // handle specific nodes
    case _ => super.traverseStm(s)
  }
}
```

For each unit of functionality such as `Arith` or `IfThenElse` the traversal actions can be defined separately as `MyTraversalArith` and `MyTraversalIfThenElse`.

Finally, we can use our traversal as follows:

```
trait Prog extends Arith {
  def main = ... // program code here
}
val impl = new Prog with ArithExp
val res = impl.reifyBlock(impl.main)
val inspect = MyTraversalArith { val IR: impl.type = impl }
inspect.traverseBlock(res)
```

### 8.2.2 Solving the "Expression Problem"

In essence, traversals confront us with the classic "expression problem" of independently extending a data model with new data variants and new operations [152]. There are many solutions to this problem but most of them are rather heavyweight. More lightweight implementations are possible in languages that support multi-methods, i.e. dispatch method calls dynamically based on the actual types of all the arguments. We can achieve essentially the same using pattern matching and mixin composition, making use of the fact that composing traits is subject to linearization [100]. We package each set of specific traversal rules into its own trait, e.g. `MyTraversalArith` that inherits from `MyTraversalBase` and overrides `traverseStm`. When the arguments do not match the rewriting pattern, the overridden method will invoke the "parent" implementation using **super**. When several such traits are combined, the super calls will traverse the overridden method implementations according to the linearization order of their containing traits. The use of pattern matching and super calls is similar to earlier work on extensible algebraic data types with defaults [160], which supported linear extensions but not composition of independent extensions.

Implementing multi-methods in a statically typed setting usually poses three problems: separate type checking/compilation, ensuring non-ambiguity and ensuring exhaustiveness. The described encoding supports separate type-checking and compilation in as far as traits do. Ambiguity is ruled out by always following the linearization order and the first-match semantics of pattern matching. Exhaustiveness is ensured at the type level by requiring a default implementation, although no guarantees can be made that the default will not choose to throw an exception at runtime. In the particular case of traversals, the default is always safe and will just continue the structural traversal.

### 8.2.3 Generating Code

Code generation is just a traversal pass that prints code. Compiling and executing code can use the same mechanism as described in Section 6.2.6.

### 8.2.4 Modularity: Adding Transformations

Transformations work very similar to traversals. One option is to traverse and transform an existing program more or less in place, not actually modifying data but attaching new Defs to existing Syms:

```
trait SimpleTransformer {
  val IR: Expressions
  import IR._
  def transformBlock[T](b: Block[T]): Block[T] =
    Block(b.stms.flatMap(transformStm), transformExp(b.res))
  def transformStm[T](s: Stm[T]): List[Stm] =
    List(Stm(s.lhs, transformDef(s.rhs)))   // preserve existing symbol s
  def transformDef[T](d: Def[T]): Def[T]    // default: use reflection
                                            // to map over case classes
```

```
}
```

An implementation is straightforward:

```
trait MySimpleTransformer extends SimpleTransformer {
  val IR: IfThenElseExp
  import IR._
  // override transformDef for each Def subclass
  def transformDef[T](d: Def[T]): Def[T] = d match {
    case IfThenElse(c,a,b) =>
      IfThenElse(transformExp(c), transformBlock(a), transformBlock(b))
    case _ => super.transformDef(d)
  }
}
```

### 8.2.5 Transformation by Iterated Staging

Another option that is more principled and in line with the idea of making compiler transforms programmable through the use of staging is to traverse the old program and create a new program. Effectively we are implementing an IR interpreter that executes staging commands, which greatly simplifies the implementation of the transform and removes the need for low-level IR manipulation.

In the implementation, we will create new symbols instead of reusing existing ones so we need to maintain a substitution that maps old to new Syms. The core implementation is given below:

```
trait ForwardTransformer extends ForwardTraversal {
  val IR: Expressions
  import IR._
  var subst: Map[Exp[_],Exp[_]]
  def transformExp[T](s: Exp[T]): Exp[T] = ... // lookup s in subst
  def transformDef[T](d: Def[T]): Exp[T]       // default
  def transformStm[T](s: Stm[T]): Exp[T] = {
    val e = transformDef(s.rhs); subst += (s.sym -> e); e
  }
  override def traverseStm[T](s: Stm[T]): Unit = {
    transformStm(s)
  }
  def reflectBlock[T](b: Block[T]): Exp[T] = withSubstScope {
    traverseBlock(b); transformExp(b.res)
  }
  def transformBlock[T](b: Block[T]): Block[T] = {
    reifyBlock(reflectBlock(b))
  }
}
```

Here is a simple identity transformer implementation for conditionals and array construction:

```
trait MyTransformer extends ForwardTransformer {
  val IR: IfThenElseExp with ArraysExp
  import IR._
  def transformDef[T](d: Def[T]): Exp[T] = d match {
    case IfThenElse(c,a,b) =>
      __ifThenElse(transformExp(c), reflectBlock(a), reflectBlock(b))
    case ArrayFill(n,i,y) =>
      arrayFill(transformExp(n), { j => withSubstScope(i -> j) { reflectBlock(y) }})
    case _ => ...
  }
}
```

The staged transformer facility can be extended slightly to translate not only within a single language but also between two languages:

```
trait FlexTransformer {
  val SRC: Expressions
  val DST: Base
  trait TypeTransform[A,B]
  var subst: Map[SRC.Exp[_],DST.Rep[_]]
  def transformExp[A,B](s: SRC.Exp[A])(implicit t: TypeTransform[A,B]): DST.Rep[B]
}
```

It is also possible to add more abstraction on top of the base transforms to build combinators for rewriting strategies in the style of Stratego [71] or Kiama [120].

## 8.3 Problem: Phase Ordering

This all works but is not completely satisfactory. With fine grained separate transformations we immediately run into phase ordering problems [149, 23]. We could execute optimization passes in a loop until we reach a fixpoint but even then we may miss opportunities if the program contains loops. For best results, optimizations need to be tightly integrated. Optimizations need a different mechanisms than lowering transformations that have a clearly defined before and after model. In the next chapter, we will thus consider a slightly different IR representation.

# Chapter 9

# Intermediate Representation: Graphs

To remedy phase ordering problems and overall allow for more flexibility in rearranging program pieces, we switch to a program representation based on structured graphs. This representation is not to be confused with control-flow graphs (CFGs): Since one of our main goals is parallelization, a sequential CFG would not be a good fit.

## 9.1 Purely Functional Subset

Let us first consider a purely functional language subset. There are much more possibilities for aggressive optimizations. We can rely on referential transparency: The value of an expression is always the same, no matter when and where it is computed. Thus, optimizations do not need to check availability or lifetimes of expressions. Global common subexpression elimination (CSE), pattern rewrites, dead code elimination (DCE) and code motion are considerably simpler than the usual implementations for imperative programs.

We switch to a "sea of nodes"-like [24] representation that is a directed (and for the moment, acyclic) graph:

```
trait Expressions {
  // expressions (atomic)
  abstract class Exp[T]
  case class Const[T](x: T) extends Exp[T]
  case class Sym[T](n: Int) extends Exp[T]
  def fresh[T]: Sym[T]

  // definitions (composite, subclasses provided by other traits)
  abstract class Def[T]

  // blocks -- no direct links to statements
  case class Block[T](res: Exp[T])

  // bind definitions to symbols automatically
  // by creating a statement
  implicit def toAtom[T](d: Def[T]): Exp[T] =
```

```
    reflectPure(d)

  def reifyBlock[T](b: =>Exp[T]): Block[T]
  def reflectPure[T](d: Def[T]): Sym[T] =
    findOrCreateDefinition(d)

  def findDefinition[T](s: Sym[T]): Option[Def[T]]
  def findDefinition[T](d: Def[T]): Option[Sym[T]]
  def findOrCreateDefinition[T](d: Def[T]): Sym[T]
}
```

It is instructive to compare the definition of trait `Expressions` with the one from the previous Chapter 8. Again there are three categories of objects involved: expressions, which are atomic (subclasses of `Exp`: constants and symbols; with a "gensym" operator `fresh` to create fresh symbols), definitions, which represent composite operations (subclasses of `Def`, to be provided by other components), and blocks, which model nested scopes.

Trait `Expressions` now provides methods to find a definition given a symbol or vice versa. Direct links between blocks and statements are removed. The actual graph nodes are `(Sym[T], Def[T])` pairs. They need not be accessible to clients at this level. Thus method `reflectStm` from the previous chapter is replaced by `reflectPure`.

Graphs also carry nesting information (boundSyms, see below). This enables code motion for different kinds of nested expressions such as lambdas, not only for loops or conditionals. The structured graph representation is also more appropriate for parallel execution than the traditional sequential control-flow graph. Pure computation can float freely in the graph and can be scheduled for execution anywhere.

### 9.1.1 Modularity: Adding IR Node Types

The object language implementation code is the same compared to the tree representation:

```
trait BaseExp extends Base with Expressions {
  type Rep[T] = Exp[T]
}
```

Again, we have separate traits, one for each unit of functionality:

```
trait ArithExp extends BaseExp with Arith {
  case class Plus(x: Exp[Double], y: Exp[Double]) extends Def[Double]
  case class Times(x: Exp[Double], y: Exp[Double]) extends Def[Double]
  def infix_+(x: Rep[Double], y: Rep[Double]) = Plus(x, y)
  def infix_*(x: Rep[Double], y: Rep[Double]) = Times(x, y)
  ...
}
trait IfThenElseExp extends BaseExp with IfThenElse {
  case class IfThenElse(c: Exp[Boolean], a: Block[T], b: Block[T]) extends Def[T]
  def __ifThenElse[T](c: Rep[Boolean], a: =>Rep[T], b: =>Rep[T]): Rep[T] =
    IfThenElse(c, reifyBlock(a), reifyBlock(b))
}
```

## 9.2 Simpler Analysis and More Flexible Transformations

Several optimizations are very simple to implement on this purely functional graph IR. The implementation draws inspiration from previous work on compiling embedded DSLs [42, 84] as well as staged FFT kernels [76].

### 9.2.1 Common Subexpression Elimination/Global Value Numbering

Common subexpressions are eliminated during IR construction using hash consing:

```
def findOrCreateDefinition[T](d: Def[T]): Sym[T]
```

Invoked by `reflectPure` through the implicit conversion method `toAtom`, this method converts a definition to an atomic expression and links it to the scope being built up by the innermost enclosing `reifyBlock` call. When the definition is known to be side-effect free, it will search the already encountered definitions for a structurally equivalent one. If a matching previous definition is found, its symbol will be returned, possibly moving the definition to a parent scope to make it accessible. If the definition may have side effects or it is seen for the first time, it will be associated with a fresh symbol and saved for future reference. This simple scheme provides a powerful global value numbering optimization [21] that effectively prevents generating duplicate code.

### 9.2.2 Pattern Rewrites

Using `findDefinition`, we can implement an extractor object [43] that enables pattern matching on a symbol to lookup the underlying definition associated to the symbol:

```
object Def {
  def unapply[T](s: Exp[T]): Option[Def[T]] = s match {
    case s: Sym[T] => findDefinition(s)
    case _ => None
  }
}
```

This extractor object can be used to implement smart constructors for IR nodes that deeply inspect their arguments:

```
def infix_*(x: Exp[Double], y: Exp[Double]) = (x,y) match {
  case (Const(x), Const(y)) => Const(x * y)
  case (Const(k), Def(Times(Const(l), y))) => Const(k * l) * y
  case _ => Times(x,y)
}
```

Smart constructors are a simple yet powerful rewriting facility. If the smart constructor is the only way to construct `Times` nodes we obtain a strong guarantee: No `Times` node is ever created without applying all possible rewrites first.

### 9.2.3 Modularity: Adding new Optimizations

Some profitable optimizations, such as the global value numbering described above, are very generic. Other optimizations apply only to specific aspects of functionality, for example particular implementations of constant folding (or more generally symbolic rewritings) such as replacing computations like `x * 1.0` with `x`. Yet other optimizations are specific to the actual program being staged. Kiselyov et al. [76] describe a number of rewritings that are particularly effective for the patterns of code generated by a staged FFT algorithm but not as much for other programs. The FFT example is discussed in more detail in Section 14.3.

What we want to achieve again is modularity, so that optimizations can be combined in a way that is most useful for a given task. To implement a particular rewriting rule (whether specific or generic), say, `x * 1.0` → `x`, we can provide a specialized implementation of `infix_*` (overriding the one in trait `ArithExp`) that will test its arguments for a particular pattern. How this can be done in a modular way is shown by the traits `ArithExpOpt` and `ArithExpOptFFT`, which implement some generic and program specific optimizations. Note that the use of `x*y` within the body of `infix_*` will apply the optimization recursively.

The appropriate pattern is to override the smart constructor in a separate trait and call the super implementation if no rewrite matches. This decouples optimizations from node type definitions.

```
trait ArithExpOpt extends ArithExp {
  override def infix_*(x:Exp[Double],y:Exp[Double]) = (x,y) match {
    case (Const(x), Const(y)) => Const(x * y)
    case (x, Const(1)) => x
    case (Const(1), y) => x
    case _ => super.infix_*(x, y)
  }
}
trait ArithExpOptFFT extends ArithExp {
  override def infix_*(x:Exp[Double],y:Exp[Double]) = (x,y) match {
    case (Const(k), Def(Times(Const(l), y))) => Const(k * l) * y
    case (x, Def(Times(Const(k), y))) => Const(k) * (x * y))
    case (Def(Times(Const(k), x)), y) => Const(k) * (x * y))
    ...
    case (x, Const(y)) => Const(y) * x
    case _ => super.infix_*(x, y)
  }
}
```

Note that the trait linearization order defines the rewriting strategy. We still maintain our guarantee that no `Times` node could be rewritten further.

Figure 9.1 shows the component architecture formed by base traits and corresponding optimizations.

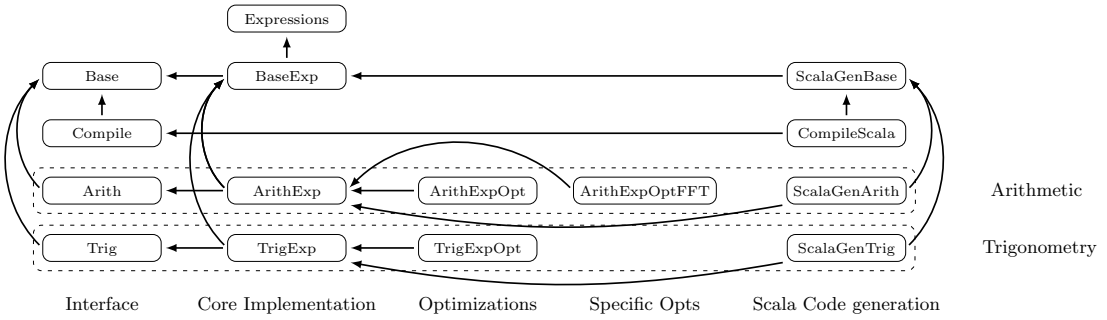



Figure 9.1: Component architecture. Arrows denote extends relationships, dashed boxes represent units of functionality.

### 9.2.4 Context- and Flow-Sensitive Transformations

Context and flow sensitive transformation become very important once we introduce effects. But even pure functional programs can profit from context information:

```
if (c) { if (c) a else b } else ...
```

The inner test on the same condition is redundant and will always succeed. How do we detect this situation? In other cases we can use the Def extractor to lookup the definition of a symbol. This will not work here, because Def works on Exp input and produces a Def object as output. We however need to work on the level of Exps, turning a Sym into `Const(true)` based on context information.

We need to adapt the way we construct IR nodes. When we enter the then branch, we add `c→Const(true)` to a substitution. This substitution needs to be applied to arguments of IR nodes constructed within the then branch.

One possible solution would be add yet another type constructor, `Ref`, with an implicit conversion from Exp to Ref that applies the substitution. A signature like `IfThenElse(c: Exp, ...)` would become `IfThenElse(c: Ref, ...)`. A simpler solution is to implement `toAtom` in such a way that it checks the resulting Def if any of its inputs need substitution and if so invoke `mirror` (see below) on the result Def, which will apply the substitution, call the appropriate smart constructor and finally call `toAtom` again with the transformed result.

### 9.2.5 Graph Transformations

In addition to optimizations performed during graph constructions, we can also implement transformation that work directly on the graph structure. This is useful if we need to do analysis on a larger portion of the program first and only then apply the transformation. An example would be to find all `FooBar` statements in a graph, and replace them uniformly with `BarBaz`. All dependent statements should re-execute their pattern rewrites, which might trigger on the new `BarBaz` input.

We introduce the concept of *mirroring*: Given an IR node, we want to apply a substitution (or generally, a `Transformer`) to the arguments and call the appropriate smart constructor again. For every IR node type we require a default `mirror` implementation that calls back its smart constructor:

```
override def mirror[A](e: Def[A], f: Transformer): Exp[A] = e match {
  case Plus(a,b) => f(a) + f(b) // calls infix_+
  case Times(a,b) => f(a) * f(b)
  case _ => super.mirror(e,f)
}
```

There are some restrictions if we are working directly on the graph level: In general we have no (or only limited) context information because a single IR node may occur multiple times in the final program. Thus, attempting to simplify effectful or otherwise context-dependent expressions will produce wrong results without an appropriate context. For pure expressions, a smart constructor called from `mirror` should not create new symbols apart from the result and it should not call reifyBlock. Otherwise, if we were creating new symbols when nothing changes, the returned symbol could not be used to check convergence of an iterative transformation easily.

The `Transfomer` argument to `mirror` can be queried to find out whether `mirror` is allowed to call context dependent methods:

```
override def mirror[A](e: Def[A], f: Transformer): Exp[A] = e match {
  case IfThenElse(c,a,b) =>
    if (f.hasContext)
      __ifThenElse(f(c),f.reflectBlock(a),f.reflectBlock(b))
    else
      ifThenElse(f(c),f(a),f(b)) // context-free version
  case _ => super.mirror(e,f)
}
```

If the context is guaranteed to be set up correctly, we call the regular smart constructor and use `f.reflectBlock` to call mirror recursively on the contents of blocks `a` and `b`. Otherwise, we call a more restricted context free method.

### 9.2.6 Dead Code Elimination

Dead code elimination can be performed purely on the graph level, simply by finding all statements reachable from the final result and discarding everything else.

We define a method to find all symbols a given object references directly:

```
def syms(x: Any): List[Sym[Any]]
```

If `x` is a Sym itself, `syms(x)` will return `List(x)`. For a case class instance that implements the `Product` interface such as `Times(a,b)`, it will return `List(a,b)` if both `a` and `b` are Syms. Since the argument type is `Any`, we can apply `syms` not only to Def objects directly but also to lists of Defs, for example.

Then, assuming `R` is the final program result, the set of remaining symbols in the graph `G` is the least fixpoint of:

```
G = R ∪ syms(G map findDefinition)
```

Dead code elimination will discard all other nodes.

## 9.3 From Graphs Back to Trees

To turn program graphs back into trees for code generation we have to decide which graph nodes should go where in the resulting program. This is the task of code motion.

### 9.3.1 Code Motion

Other optimizations can apply transformations optimistically and need not worry about maintaining a correct schedule: Code motion will fix it up. The algorithm will try to push statements inside conditional branches and hoist statements out of loops. Code motion depends on dependency and frequency information but not directly on data-flow information. Thus it can treat functions or other user defined compound statements in the same way as loops. This makes our algorithm different from code motion algorithms based on data flow analysis such as Lazy Code Motion (LCM, [77]) or Partial Redundancy Elimination (PRE, [74]).

The graph IR reflects "must before" (ordering) and "must inside" (containment) relations, as well as anti-dependence and frequency. These relations are implemented by the following methods, which can be overridden for new definition classes:

```
def syms(e: Any): List[Sym[Any]]        // value dependence (must before)
def softSyms(e: Any): List[Sym[Any]]    // anti dependence (must not after)
def boundSyms(e: Any): List[Sym[Any]]   // nesting dependence (must not outside)
def symsFreq(e: Any): List[(Sym[Any],   // frequency information (classify
  Double)]                              // sym as 'hot', 'normal', 'cold')
```

To give an example, `boundSyms` applied to a loop node `RangeForeach(range,idx,body)` with index variable `idx` would return `List(idx)` to denote that `idx` is fixed "inside" the loop expression.

Given a subgraph and a list of result nodes, the goal is to identify the graph nodes that should form the "current" level, as opposed to those that should remain in some "inner" scope, to be scheduled later. We will reason about the paths on which statements can be reached from the result. The first idea is to retain all nodes on the current level that are reachable on a path that does not cross any conditionals, i.e. that has no "cold" refs. Nodes only used from conditionals will be pushed down. However, this approach does not yet reflect the precedence of loops. If a loop is top-level, then conditionals inside the loop (even if deeply nested) should not prevent hoisting of statements. So we refine the characterization to retain all nodes that are reachable on a path that does not cross top-level conditionals.

This leads to a simple iterative algorithm (see Figure 9.3): Starting with the known top level statements, nodes reachable via normal links are added and for each hot ref, we follow nodes that are forced inside until we reach one that can become top-level again.

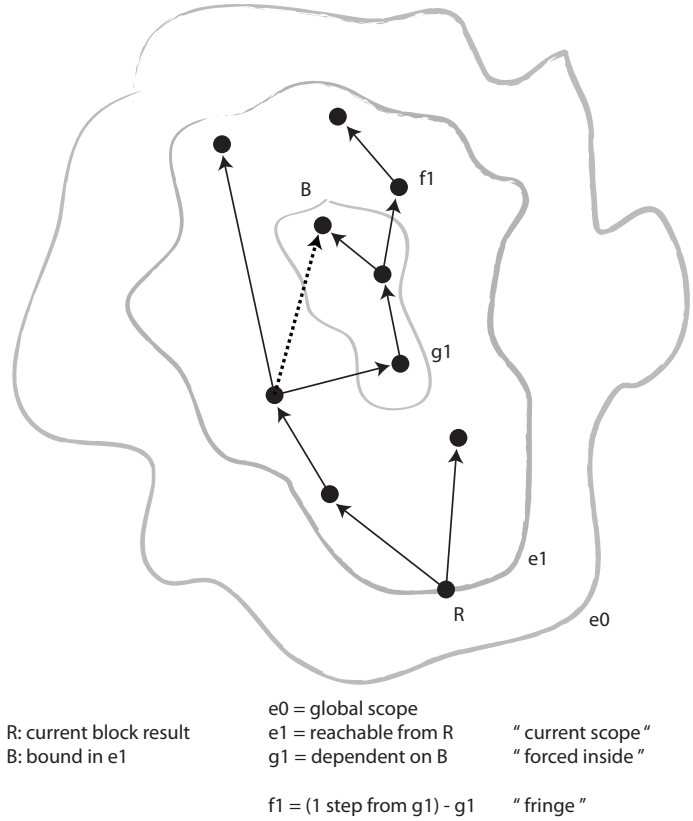



Figure 9.2: Graph IR with regular and nesting edges (boundSyms, dotted line) as used for code motion.

Code Motion Algorithm: Compute the set $L$ of top level statements for the current block, from a set of available statements $E$, a set of forced-inside statements $G \subseteq E$ and a block result $R$.

1. Start with $L$ containing the known top level statements, initially the (available) block result $R \cap E$.
2. Add to $L$ all nodes reachable from $L$ via normal links (neither hot nor cold) through $E - G$ (not forced inside).
3. For each hot ref from $L$ to a statement in $E - L$, follow any links through $G$, i.e. the nodes that are forced inside, if there are any. The first non-forced-inside nodes (the "hot fringe") become top level as well (add to $L$).
4. Continue with 2 until a fixpoint is reached.

Figure 9.3: Code Motion algorithm.

To implement this algorithm, we need to determine the set G of nodes that are forced inside and may not be part of the top level. We start with the block result R and a graph E that has all unnecessary nodes removed (DCE already performed):

```
E = R ∪ syms(E map findDefinition)
```

We then need a way to find all uses of a given symbol s, up to but not including the node where the symbol is bound:

```
U(s) = {s} ∪ { g ∈ E | syms(findDefinition(g)) ∩ U(s) ≠ ∅
                     && s ∉ boundSyms(findDefinition(g))) }
```

We collect all bound symbols and their dependencies. These cannot live on the current level, they are forced inside:

```
B = boundSyms (E map findDefinition)
G = union (B map U)    // must inside
```

Computing U(s) for many symbols s individually is costly but implementations can exploit considerable sharing to optimize the computation of G.

The iteration in Figure 9.3 uses G to follow forced-inside nodes after a hot ref until a node is found that can be moved to the top level.

Let us consider a few examples to build some intuition about the code motion behavior. In the code below, the starred conditional is on the fringe (first statement that can be outside) and on a hot path (through the loop). Thus it will be hoisted. Statement foo will be moved inside:

```
loop { i =>                  z = *if (x) foo
  if (i > 0)                 loop { i =>
    *if (x)                    if (i > 0)
      foo                        z
}                            }
```

The situation changes if the inner conditional is forced inside by a value dependency. Now statement foo is on the hot fringe and becomes top level.

```
loop { i =>                  z = *foo
  if (x)                     loop { i =>
    if (i > 0)                 if (x)
      *foo                       if (i > 0)
}                                  z
                             }
```

For loops inside conditionals, the containing statements will be moved inside (relative to the current level).

```
if (x)                       if (x)
  loop { i =>                  z = foo
    foo                        loop { i =>
  }                              z
                               }
```

#### Pathological Cases

The described algorithm works well and is reasonably efficient in practice. Being a heuristic, it cannot be optimal in all cases. Future versions could employ more elaborate cost models instead of the simple hot/cold distinction. One case worth mentioning is when a statement is used only in conditionals but in different conditionals:

```
z = foo                    if (x)
if (x)                       foo
  z                        if (y)
if (y)                       foo
  z
```

In this situation `foo` will be duplicated. Often this duplication is beneficial because `foo` can be optimized together with other statements inside the branches. In general of course there is a danger of slowing down the program if both conditions are likely to be true at the same time. In that case it would be a good idea anyways to restructure the program to factor out the common criteria into a separate test.

#### Scheduling

Once we have determined which statements should occur on which level, we have to come up with an ordering for the statements. Before starting the code motion algorithm, we sort the input graph in topological order and we will use the same order for the final result. For the purpose of sorting, we include anti-dependencies in the topological sort although they are disregarded during dead code elimination. A bit of care must be taken though: If we introduce loops or recursive functions the graph can be cyclic, in which case no topological order exists. However, cycles are caused only by inner nodes pointing back to outer nodes and for sorting purposes we can remove these back-edges to obtain an acyclic graph.

### 9.3.2 Tree-Like Traversals and Transformers

To generate code or to perform transformation by iterated staging (see Section 8.2.5) we need to turn our graph back into a tree. The interface to code motion allows us to build a generic tree-like traversal over our graph structure:

```
trait Traversal {
  val IR: Expressions; import IR._
  // perform code motion, maintaining current scope
  def focusExactScope(r: Exp[Any])(body: List[Stm[Any]] => A): A
  // client interface
  def traverseBlock[T](b: Block[T]): Unit =
    focusExactScope(b.res) { levelScope =>
      levelScope.foreach(traverseStm)
    }
  def traverseStm[T](s: Stm[T]): Unit = blocks(s).foreach(traverseBlock)
}
```

This is useful for other analyses as well, but in particular for building transformers that traverse one graph in a tree like fashion and create another graph analogous to Section 8.2.5. The implementation of trait `ForwardTransformer` carries over almost unmodified.

## 9.4 Effects

To ensure that operations can be safely moved around (and for other optimizations as well), a compiler needs to reason about their possible side effects. The graph representation presented so far is pure and does not mention effects at all. However all the necessary ingredients are already there: We can keep track of side effects simply by making effect dependencies explicit in the graph. In essence, we turn all programs into functional programs by adding an invisible state parameter (similar in spirit but not identical to SSA conversion).

### 9.4.1 Simple Effect Domain

We first consider global effects like console output via `println`. Distinguishing only between "has no effect" and "may have effect" means that all operations on mutable data structures, including reads, have to be serialized along with all other side effects.

By default, we assume operations to be pure (i.e. side-effect free). Programmers can designate effectful operations by using `reflectEffect` instead of the implicit conversion `toAtom` which internally delegates to `reflectPure`. Console output, for example, is implemented like this:

```
def print(x: Exp[String]): Exp[Unit] = reflectEffect(Print(x))
```

The call to `reflectEffect` adds the passed IR node to a list of effects for the current block. Effectful expressions will attract dependency edges between them to ensure serialization. A compound expression such as a loop or a conditional will internally use `reifyBlock`, which attaches nesting edges to the effectful nodes contained in the block.

Internally, `reflectEffect` creates `Reflect` nodes that keep track of the context dependencies:

```
var context: List[Exp[Any]]
case class Reflect[T](d: Def[T], es: List[Sym[Any]]) extends Def[T]
def reflectEffect[T](d: Def[T]): Exp[T] = createDefinition(Reflect(d, context)).sym
```

The context denotes the "current state". Since state can be seen as an abstraction of effect history, we just define context as a list of the previous effects.

In this simple model, all effect dependencies are uniformly encoded in the IR graph. Rewriting, CSE, DCE, and Code Motion are disabled for effectful statements (very pessimistic). Naturally we would like something more fine grained for mutable data.

### 9.4.2 Fine Grained Effects: Tracking Mutations per Allocation Site

We can add other, more fine grained, variants of `reflectEffect` which allow tracking mutations per allocation site or other, more general abstractions of the heap that provide a

partitioning into regions. Aliasing and sharing of heap objects such as arrays can be tracked via optional annotations on IR nodes. Reads and writes of mutable objects are automatically serialized and appropriate dependencies inserted to guarantee a legal execution schedule.

Effectful statements are tagged with an effect summary that further describes the effect. The summary can be extracted via `summarizeEffects`, and there are some operations on summaries (like `orElse`, `andThen`) to combine effects. As an example consider the definition of conditionals, which computes the compound effect from the effects of the two branches:

```
def __ifThenElse[T](cond: Exp[Boolean], thenp: => Rep[T], elsep: => Rep[T]) {
  val a = reifyBlock(thenp)
  val b = reifyBlock(elsep)
  val ae = summarizeEffects(a) // get summaries of the branches
  val be = summarizeEffects(b)
  val summary = ae orElse be   // compute summary for whole expression
  reflectEffect(IfThenElse(cond, a, b), summary)  // reflect compound expression
                                                  // (effect might be none, i.e. pure)
}
```

To specify effects more precisely for different kinds of IR nodes, we add further `reflect` methods:

```
reflectSimple     // a 'simple' effect: serialized with other simple effects
reflectMutable    // an allocation of a mutable object; result guaranteed unique
reflectWrite(v)   // a write to v: must refer to a mutable allocation
                  // (reflectMutable IR node)
reflectRead(v)    // a read of allocation v (not used by programmer,
                  // inserted implicitly)
reflectEffect(s)  // provide explicit summary s, specify may/must info for
                  // multiple reads/writes
```

The framework will serialize reads and writes so to respect data and anti-dependency with respect to the referenced allocations. To make this work we also need to keep track of sharing and aliasing. Programmers can provide for their IR nodes a list of input expressions which the result of the IR node may alias, contain, extract from or copy from.

```
def aliasSyms(e: Any): List[Sym[Any]]
def containSyms(e: Any): List[Sym[Any]]
def extractSyms(e: Any): List[Sym[Any]]
def copySyms(e: Any): List[Sym[Any]]
```

These four pieces of information correspond to the possible pointer operations `x = y`, `*x = y`, `x = *y` and `*x = *y`. Assuming an operation `y = Foo(x)`, x should be returned in the following cases:

```
x ∈ aliasSyms(y)     if y = x      // if then else
x ∈ containSyms(y)   if *y = x     // array update
x ∈ extractSyms(y)   if y = *x     // array apply
x ∈ copySyms(y)      if *y = *x    // array clone
```

Here, `y = x` is understood as "y may be equal to x", `*y = x` as "dereferencing y (at some index) may return x" etc.

### Restricting Possible Effects

It is often useful to restrict the allowed effects somewhat to make analysis more tractable and provide better optimizations. One model, which works reasonably well for many applications, is to prohibit sharing and aliasing between mutable objects. Furthermore, read and write operations must unambiguously identify the allocation site of the object being accessed. The framework uses the aliasing and points-to information to enforce these rules and to keep track of immutable objects that point to mutable data. This is to make sure the right serialization dependencies and `reflectRead` calls are inserted for operations that may reference mutable state in an indirect way.

# Chapter 10

# Advanced Optimizations

We have seen above in Chaper 9 how many classic compiler optimizations can be applied to the IR generated from embedded programs in a straightforward way. Due to the structure of the IR, these optimizations all operate in an essentially global way, at the level of domain operations. In this chapter we discuss some other advanced optimizations that can be implemented on the graph IR. We present more elaborate examples for how these optimizations benefit larger use cases later in Part III.

## 10.1 Rewriting

Many optimizations that are traditionally implemented using an iterative dataflow analysis followed by a transformation pass can also be expressed using various flavors of rewriting. Whenever possible we tend to prefer the rewriting version because rewrite rules are easy to specify separately and do not require programmers to define abstract interpretation lattices.

### 10.1.1 Context-Sensitive Rewriting

Smart constructors in our graph IR can be context sensitive. For example, reads of local variables examine the current effect context to find the last assignment, implementing a form of copy propagation (middle):

```
var x = 7      var x = 7     println(5)
x = 5          x = 5
println(x)     println(5)
```

This renders the stores dead, and they will be removed by dead code elimination later (right).

### 10.1.2 Speculative Rewriting: Combining Analyses and Transformations

Many optimizations are mutually beneficial. In the presence of loops, optimizations need to make optimistic assumptions for the supporting analysis to obtain best results. If multiple analyses are run separately, each of them effectively makes pessimistic assumptions about the outcome of all others. Combined analyses avoid the phase ordering problem by solving

everything at the same time. Lerner, Grove, and Chambers showed a method of composing optimizations by interleaving analyses and transformations [85]. We use a modified version of their algorithm that works on structured loops instead of CFGs and using dependency information and rewriting instead of explicit data flow lattices. Usually, rewriting is semantics preserving, i.e. pessimistic. The idea is to drop that assumption. As a corollary, we need to rewrite speculatively and be able to rollback to a previous state to get optimistic optimization. The algorithm proceeds as follows: for each encountered loop, apply all possible transforms to the loop body, given empty initial assumptions. Analyze the result of the transformation: if any new information is discovered throw away the transformed loop body and retransform the original with updated assumptions. Repeat until the analysis result has reached a fixpoint and keep the last transformation as result.

Here is an example of speculative rewriting, showing the initial optimistic iteration (middle), with the fixpoint (right) reached after the second iteration:

```
var x = 7                  var x = 7          var x = 7 //dead
var c = 0                  var c = 0          var c = 0
while (c < 10) {           while (true) {     while (c < 10) {
  if (x < 10) print("!")     print("!")         print("!")
  else x = c                 print(7)           print(7)
  print(x)                   print(0)           print(c)
  print(c)                   c = 1              c += 1
  c += 1                   }                  }
}
```

This algorithm allows us to do all forward data flow analyses and transforms in one uniform, combined pass driven by rewriting. In the example above, during the initial iteration (middle), separately specified rewrites for variables and conditionals work together to determine that x=c is never executed. At the end of the loop body we discover the write to c, which invalidates our initial optimistic assumption c=0. We rewrite the original body again with the augmented information (right). This time there is no additional knowledge discovered so the last speculative rewrite becomes the final one.

### 10.1.3 Delayed Rewriting and Multi-Level IR

For some transformations, e.g. data structure representation lowering, we do not execute rewrites now, but later, to give further immediate rewrites a change to match on the current expression before it is rewritten. This is a simple form of prioritizing different rewrites, in this case optimizations over lowerings. It also happens to be a central idea behind telescoping languages [73].

We perform simplifications eagerly, after each transform phase. Thus we guarantee that CSE, DCE etc. have been applied on high-level operations before they are translated into lower-level equivalents, on which optimizations would be much harder to apply.

We call the mechanism to express this form of rewrites *delayed* rewriting. Here is an example that delayedly transforms a plus operation on Vectors into an operation on arrays:

```
def infix_plus(a: Rep[Vector[Double]], b: Rep[Vector[Double]]) = {
```

```
  VectorPlus(a,b) atPhase(lowering) {
    val data = Array.fill(a.length) { i => a(i) + b(i) }
    vector_with_backing_array(data)
  }
}
```

The transformation is only carried out at phase `lowering`. Before that, the IR node remains a `VectorPlus` node, which allows other smart constructor rewrites to kick in that match their arguments against `VectorPlus`.

Technically, delayed rewriting is implemented using a worklist transformer that keeps track of the rewrites to be performed during the next iteration. The added convenience over using a transformer directly is that programmers can define simple lowerings inline without needing to subclass and install a transformer trait.

## 10.2 Splitting and Combining Statements

Since our graph IR contains structured expressions, optimizations need to work with compound statements. Reasoning about compound statements is not easy: For example, our simple dead code elimination algorithm will not be able to remove only pieces of a compound expression. Our solution is simple yet effective: We eagerly split many kinds of compound statements, assuming optimistically that only parts will be needed. We find out which parts through the regular DCE algorithm. Afterwards we reassemble the remaining pieces.

### 10.2.1 Effectful Statements

A good example of statement splitting are effectful conditionals:

```
var a, b, c = ...
if (cond) {
  a = 9
  b = 1
} else
  c = 3
println(a+c)
```

```
var a, b, c = ...
if (cond)
  a = 9
if (cond)
  b = 1
if (!cond)
  c = 3
println(a+c)
```

```
var a, c = ...
if (cond)
  a = 9
else
  c = 3
println(a+c)
```

From the conditional in the initial program (left), splitting creates three separate expressions, one for each referenced variable (middle). Pattern rewrites are executed when building the split nodes but do not have any effect here. Dead code elimination removes the middle one because variable b is not used, and the remaining conditionals are merged back together (right). Of course successful merging requires to keep track of how expressions have been split.

### 10.2.2 Data Structures

Splitting is also very effective for data structures, as often only parts of a data structure are used or modified. We can define a generic framework for data structures:

```
trait StructExp extends BaseExp {
  abstract class StructTag
  case class Struct[T](tag: StructTag, elems: Map[String,Rep[Any]]) extends Def[T]
  case class Field[T](struct: Rep[Any], key: String) extends Def[T]

  def struct[T](tag: StructTag, elems: Map[String,Rep[Any]]) = Struct(tag, elems)
  def field[T](struct: Rep[Any], key: String): Rep[T] = struct match {
    case Def(Struct(tag, elems)) => elems(index).asInstanceOf[Rep[T]]
    case _ => Field[T](struct, index)
  }
}
```

There are two IR node types, one for structure creation and one for field access. The structure creation node contains a hash map that holds (static) field identifiers and (dynamic) field values. It also contains a `tag` that can be used to hold further information about the nature of the data structure. The interface for field accesses is method `field`, which pattern matches on its argument and, if that is a `Struct` creation, looks up the desired value from the embedded hash map.

We continue by adding a rule that makes the result of a conditional a `Struct` if the branches return `Struct`:

```
override def ifThenElse[T](cond: Rep[Boolean], a: Rep[T], b: Rep[T]) =
(a,b) match {
  case (Def(Struct(tagA,elemsA)), Def(Struct(tagB, elemsB))) =>
    assert(tagA == tagB)
    assert(elemsA.keySet == elemsB.keySet)
    Struct(tagA, elemsA.keySet map (k => ifThenElse(cond, elemsA(k), elemsB(k))))
  case _ => super.ifThenElse(cond,a,b)
}
```

Similar rules are added for many of the other core IR node types. DCE can remove individual elements of the data structure that are never used. During code generation and tree traversals, the remaining parts of the split conditional are merged back together.

We will study examples of this struct abstraction in Section 13.2 and an extension to unions and inheritance in Section 13.4.

### 10.2.3 Representation Conversion

A natural extension of this mechanism is a generic array-of-struct to struct-of-array transform. The definition is analogous to that of conditionals. We override the array constructor `arrayFill` that represents expressions of the form `Array.fill(n) { i => body }` to create a struct with an array for each component of the body if the body itself is a Struct:

```
override def arrayFill[T](size: Exp[Int], v: Sym[Int], body: Def[T]) = body match {
  case Block(Def(Struct(tag, elems))) =>
    struct[T](ArraySoaTag(tag,size),
      elems.map(p => (p._1, arrayFill(size, v, Block(p._2)))))
  case _ => super.arrayFill(size, v, body)
```

```
}
```

Note that we tag the result struct with an `ArraySoaTag` to keep track of the transformation. This class is defined as follows:

```
case class ArraySoaTag(base: StructTag, len: Exp[Int]) extends StructTag
```

We also override the methods that are used to access array elements and return the length of an array to do the right thing for transformed arrays:

```
override def infix_apply[T](a: Rep[Array[T]], i: Rep[Int]) = a match {
  case Def(Struct(ArraySoaTag(tag,len),elems)) =>
    struct[T](tag, elems.map(p => (p._1, infix_apply(p._2, i))))
  case _ => super.infix_at(a,i)
}
override def infix_length[T](a: Rep[Array[T]]): Rep[Int] = a match {
  case Def(Struct(ArraySoaTag(tag, len), elems)) => len
  case _ => super.infix_length(a)
}
```

Examples for this struct of array transformation are shown in Section 13.5 and Chapter 14.

## 10.3 Loop Fusion and Deforestation

The use of independent and freely composable traversal operations such as `v.map(..).sum` is preferable to explicitly coded loops. However, naive implementations of these operations would be expensive and entail lots of intermediate data structures. We provide a novel loop fusion algorithm for data parallel loops and traversals (see Chapter 14 for examples of use). The core loop abstraction is

`loop(s)` $\overline{\texttt{x} = \mathscr{G}}$ `{ i =>` $\overline{E[\texttt{x} \leftarrow \texttt{f(i)}]}$ `}`

where `s` is the size of the loop and `i` the loop variable ranging over $[0, \texttt{s})$. A loop can compute multiple results $\overline{\texttt{x}}$, each of which is associated with a generator $\mathscr{G}$, one of `Collect`, which creates a flat array-like data structure, `Reduce(`$\oplus$`)`, which reduces values with the associative operation $\oplus$, or `Bucket(`$\mathscr{G}$`)`, which creates a nested data structure, grouping generated values by key and applying $\mathscr{G}$ to those with matching key. Loop bodies consist of yield statements `x←f(i)` that define values passed to generators (of this loop or an outer loop), embedded in some outer context $E[.]$ that might consist of other loops or conditionals. For `Bucket` generators yield takes (key,value) pairs.

The fusion rules are summarized in Figure 10.1. This model is expressive enough to model many common collection operations:

```
x=v.map(f)     loop(v.size) x=Collect { i => x ← f(v(i)) }
x=v.sum        loop(v.size) x=Reduce(+) { i =>  x ← v(i) }
x=v.filter(p)  loop(v.size) x=Collect { i => if (p(v(i)))
                                               x ← v(i) }
x=v.flatMap(f) loop(v.size) x=Collect { i => val w = f(v(i))
                         loop(w.size) { j => x ← w(j) }}
```

Generator kinds: $\mathcal{G}$ ::= `Collect` | `Reduce(`$\oplus$`)` | `Bucket(`$\mathcal{G}$`)`
Yield statement: xs ← x
Contexts: $E$[.] ::= loops and conditionals

*Horizontal case (for all types of generators):*

```
  loop(s) x1=G1 { i1 => E1[ x1 ← f1(i1) ] }
  loop(s) y1=G2 { i2 => E2[ x2 ← f2(i2) ] }
```

---

```
loop(s) x1=G1, x2=G2 { i =>
        E1[ x1 ← f1(i) ]; E2[ x2 ← f2(i) ] }
```

*Vertical case (consume collect):*

```
   loop(s) x1=Collect { i1 => E1[ x1 ← f1(i1) ] }
loop(x1.size) x2=G { i2 => E2[ x2 ← f2(x1(i2)) ] }
```

---

```
loop(s) x1=Collect, x2=G { i =>
           E1[ x1 ← f1(i); E2[ x2 ← f2(f1(i)) ]] }
```

*Vertical case (consume bucket collect):*

```
        loop(s) x1=Bucket(Collect) { i1 =>
           E1[ x1 ← (k1(i1), f1(i1)) ] }
  loop(x1.size) x2=Collect { i2 =>
    loop(x1(i2).size) y=G { j =>
      E2[ y ← f2(x1(i2)(j)) ] }; x2 ← y }
```

---

```
loop(s) x1=Bucket(Collect), x2=Bucket(G) { i =>
    E1[ x1 ← (k1(i), f1(i));
        E2[ x2 ← (k1(i), f2(f1(i))) ]] }
```

Figure 10.1: Loop fusion

```
x=v.distinct   loop(v.size) x=Bucket(Reduce(rhs)) { i =>
                                        x ← (v(i), v(i)) }
```

Other operations are accommodated by generalizing slightly. Instead of implementing a `groupBy` operation that returns a sequence of (Key, Seq[Value]) pairs we can return the keys and values in separate data structures. The equivalent of `(ks,vs)=v.groupBy(k).unzip` is:

```
loop(v.size) ks=Bucket(Reduce(rhs)),vs=Bucket(Collect) { i =>
  ks ← (v(i), v(i)); vs ← (v(i), v(i)) }
```

In Figure 10.1, multiple instances of `f1(i)` are subject to CSE and not evaluated twice. Substituting `x1(i2)` with `f1(i)` will remove a reference to `x1`. If `x1` is not used anywhere else, it will also be subject to DCE. Within fused loop bodies, unifying index variable `i` and substituting references will trigger the uniform forward transformation pass. Thus, fusion not only removes intermediate data structures but also provides additional optimization opportunities inside fused loop bodies (including fusion of nested loops).

Fixed size array construction `Array(a,b,c)` can be expressed as

```
loop(3) x=Collect { case 0 => x ← a
                    case 1 => x ← b case 2 => x ← c }
```

and concatenation `xs ++ ys` as `Array(xs,ys).flatMap(i=>i)`:

```
loop(2) x=Collect { case 0 => loop(xs.size) { i => x ← xs(i) }
                    case 1 => loop(ys.size) { i => x ← ys(i) }}
```

Fusing these patterns with a consumer will duplicate the consumer code into each match case. Implementations should have some kind of cutoff value to prevent code explosion. Code generation does not need to emit actual loops for fixed array constructions but can just produce the right sequencing of yield operations.

Examples for the fusion algorithm are shown in Section 13.6 and Chapter 14.

# Part III

# Staging and Embedded Compilers at Work

# Chapter 11

# Intro: Abstraction Without Regret

LMS is a dynamic multi-stage programming approach: We have the full Scala language at our disposal to compose fragments of object code. In fact, DSL programs are program *generators* that produce an object program IR when run. DSL or library authors and application programmers can exploit this multi-level nature to perform computations explicitly at staging time, so that the generated program does not pay a runtime cost. Multi-stage programming shares some similarities with partial evaluation [65], but instead of an automatic binding-time analysis, the programmer makes binding times explicit in the program. We have seen how LMS uses `Rep` types for this purpose:

```
val s: Int = ...      // a static value: computed at staging time
val d: Rep[Int] = ... // a dynamic value: computed when generated program is run
```

Unlike with automatic partial evaluation, the programmer obtains a guarantee about which expressions will be evaluated at staging time.

While moving *computations* from run time to staging time is an interesting possibility, many computations actually depend on dynamic input and cannot be done before the input is available. Nonetheless, explicit staging can be used to *combine* dynamic computations more efficiently. Modern programming languages provide indispensable constructs for abstracting and combining program functionality. Without higher-order features such as first-class functions or object and module systems, software development at scale would not be possible. However, these abstraction mechanisms have a cost and make it much harder for the compiler to generate efficient code.

Using explicit staging, we can use abstraction in the generator stage to remove abstraction in the generated program. This holds both for control (e.g. functions, continuations) and data abstractions (e.g. objects, boxing). Some of the material in this chapter is taken from [113].

## 11.1 Common Compiler Optimizations

We have seen in Part II how many classic compiler optimizations can be applied to the IR generated from embedded programs in a straightforward way. Among those generic optimizations are common subexpression elimination, dead code elimination, constant folding and code

motion. Due to the structure of the IR, these optimizations all operate in an essentially global way, at the level of domain operations. An important difference to regular general-purpose compilers is that IR nodes carry information about effects they incur (see Section 9.4). This permits to use quite precise dependency tracking that provides the code generator with a lot of freedom to group and rearrange operations. Consequently, optimizations like common subexpression elimination and dead code elimination will easily remove complex DSL operations that contain internal control-flow and may span many lines of source code.

Common subexpression elimination (CSE) / global value numbering (GVN) for pure nodes is handled by `toAtom`: whenever the `Def` in question has been encountered before, its existing symbol is returned instead of a new one (see Section 9.2.1). Since the operation is pure, we do not need to check via data flow analysis whether its result is available on the current path. Instead we just insert a dependency and let the later code motion pass (see Section 9.3.1) schedule the operation in a correct order. Thus, we achieve a similar effect as partial redundancy elimination (PRE [74]) but in a simpler way.

Based on frequency information for block expression, code motion will hoist computation out of loops and push computation into conditional branches. Dead code elimination is trivially included. Both optimizations are coarse grained and work on the level of domain operations. For example, whole data parallel loops will happily be hoisted out of other loops.

Consider the following user-written code:

```
v1 map { x =>
  val s = sum(v2.length) { i => v2(i) }
  x/s
}
```

This snippet scales elements in a vector `v1` relative to the sum of `v2`'s elements. Without any extra work, the generic code motion transform places the calculation of `s` (which is itself a loop) outside the loop over `v1` because it does not depend on the loop variable `x`.

```
val s = sum(v2.length) { i => v2(i) }
v1 map { x =>
  x/s
}
```

## 11.2 Delite: An End-to-End System for Embedded Parallel DSLs

This section gives an overview of our approach to developing and executing embedded DSLs in parallel and on heterogeneous devices. A more thorough description of Delite can be found in Section 16 of Part IV.

Delite seeks to alleviate the burden of building a high performance DSL by providing reusable infrastructure. Delite DSLs are embedded in Scala using LMS. On top of this layer, Delite is structured into a *framework* and a *runtime* component. The framework provides primitives for parallel operations such as `map` or `reduce` that DSL authors can use to define higher-level operations. Once a DSL author uses Delite operations, Delite handles code

generating to multiple platforms (e.g. Scala and CUDA), and handles difficult but common issues such as device communication and synchronization. These capabilities are enabled by exploiting the domain-specific knowledge and restricted semantics of the DSL compiler.

### 11.2.1 Building a Simple DSL

On the surface, DSLs implemented on top of Delite appear very similar to purely-embedded (i.e. library only) Scala-based DSLs. However, a key aspect of LMS and hence Delite is that DSLs are split in two parts, *interface* and *implementation*. Both parts can be assembled from components in the form of Scala traits. DSL programs are written in terms of the DSL interface, agnostic of the implementation. Part of each DSL interface is an abstract type constructor `Rep[_]` that is used to wrap types in DSL programs. For example, DSL programs use `Rep[Int]` wherever a regular program would use `Int`. The DSL operations defined in the DSL interface (most of them are abstract methods) are all expressed in terms of `Rep` types.

The DSL *implementation* provides a concrete instantiation of `Rep` as expression trees (or graphs). The DSL operations left abstract in the interface are implemented to create an expression representation of the operation. Thus, as a result of executing the DSL program, we obtain an analyzable representation of the very DSL program which we will refer to as IR (intermediate representation).

To substantiate the description, let us consider an example step by step. A simple (and rather pointless) program that calculates the average of 100 random numbers, written in a prototypical DSL `MyDSL` that includes numeric vectors and basic console IO could look like this:

```
object HelloWorldRunner extends MyDSLApplicationRunner with HelloWorld
trait HelloWorld extends MyDSLApplication {
  def main() = {
    val v = Vector.rand(100)
    println("today's lucky number is: ")
    println(v.avg)
  }
}
```

Programs in our sample DSL live within traits that inherit from `MyDSLApplication`, with method `main` as the entry point.

`MyDSLApplication` is a trait provided by the DSL that defines the DSL interface. In addition to the actual DSL program, there is a singleton object that inherits from `MyDSLApplicationRunner` and mixes in the trait that contains the program. As the name implies, this object will be responsible for directing the staged execution of the DSL application.

Here is the definition of `MyDSL`'s components encountered so far:

```
trait MyDSLApplication extends DeliteApplication with MyDSL
trait MyDSLApplicationRunner extends DeliteApplicationRunner with MyDSLExp

trait MyDSL extends ScalaOpsPkg with VectorOps
trait MyDSLExp extends ScalaOpsPkgExp with VectorOpsExp with MyDSL
```

MyDSLApplicationRunner inherits the mechanics for invoking code generation from DeliteApplication. We discuss how Delite provides these facilities in section 11.2.3. We observe a structural split in the inheritance hierarchy that is rather fundamental: MyDSL defines the DSL *interface*, MyDSLExp the *implementation*. A DSL program is written with respect to the interface but it knows nothing about the implementation. The main reason for this separation is safety. If a DSL program could observe its own structure, optimizing rewrites that maintain semantic but not structural equality of DSL expressions could no longer be applied safely.[1] Our sample DSL includes a set of common Scala operations that are provided by the core LMS library as trait ScalaOpsPkg. These operations include conditionals, loops, variables and also println. On top of this set of generic things that are inherited from Scala, the DSL contains vectors and associated operations. The corresponding interface is defined as follows:

```
trait VectorOps extends Base {
  abstract class Vector[T]                    // placeholder ("phantom") type
  object Vector {
    def rand(n: Rep[Int]) = vector_rand(n)    // invoked as: Vector.rand(n)
  }
  def vector_rand(n: Rep[Int]): Rep[Vector[Double]]
  def infix_length[T](v: Rep[Vector[T]]): Rep[Int]      // invoked as: v.length
  def infix_sum[T:Numeric](v: Rep[Vector[T]]): Rep[T]   // invoked as: v.sum
  def infix_avg[T:Numeric](v: Rep[Vector[T]]): Rep[T]   // invoked as: v.avg
  ...
}
```

There is an abstract class Vector[T] for vectors with element type T. The notation T:Numeric means that T may only range over numeric types such as Int or Double. Operations on vectors are not declared as instance methods of Vector[T] but as external functions over values of type Rep[Vector[T]].

Returning to our sample DSL, this is the definition of VectorOpsExp, the implementation counterpart to the interface defined above in VectorOps:

```
trait VectorOpsExp extends DeliteOpsExp with VectorOps {
  case class VectorRand[T](n: Exp[Int]) extends Def[Vector[Double]]
  case class VectorLength[T](v: Exp[Vector[T]]) extends Def[Int]

  case class VectorSum[T:Numeric](v: Exp[Vector[T]]) extends DeliteOpLoop[Exp[T]] {
    val range = v.length
    val body = DeliteReduceElem[T](v)(_ + _) // scalar addition (impl not shown)
  }

  def vector_rand(n: Rep[Int]) = VectorRand(n)
  def infix_length[T](v: Rep[Vector[T]]) = VectorLength(v)
  def infix_sum[T:Numeric](v: Rep[Vector[T]]) = VectorSum(v)
  def infix_avg[T:Numeric](v: Rep[Vector[T]]) = v.sum / v.length
  ...
}
```

[1] In fact, this is the main reason why MSP languages do not allow inspection of staged code at all [131].

The constructor `rand` and the function `length` are implemented as new plain IR nodes (extending `Def`). Operation `avg` is implemented directly in terms of `sum` and `length` whereas `sum` is implemented as a `DeliteOpLoop` with a `DeliteReduceElem` body. These special classes of structured IR nodes are provided by the Delite framework and are inherited via `DeliteOpsExp`.

### 11.2.2 Code Generation

The LMS framework provides a code generation infrastructure that includes a program scheduler and a set of base code generators. The program scheduler uses the data and control dependencies encoded by IR nodes to determine the sequence of nodes that should be generated to produce the result of a block. After the scheduler has determined a schedule, it invokes the code generator on each node in turn. There is one *code generator* object per target platform (e.g. Scala, CUDA, C++) that mixes together traits that describe how to generate platform-specific code for each IR node. This organization makes it easy for DSL authors to modularly extend the base code generators; they only have to define additional traits to be mixed in with the base generator.

Therefore, DSL designers only have to add code generators for their own domain-specific types. They inherit the common functionality of scheduling and callbacks to the generation methods, and can also build on top of code generator traits that have already been defined. In many cases, though, DSL authors do not have to write code generators at all; the next section describes how Delite takes over this responsibility for most operations.

### 11.2.3 The Delite Compiler Framework and Runtime

On top of the LMS framework that provides the basic means to construct IR nodes for DSL operations, the Delite Compiler Framework provides high-level representations of execution patterns through `DeliteOp` IR, which includes a set of common parallel execution patterns (e.g. map, zipWith, reduce).

`DeliteOp` extends `Def`, and DSL operations may extend one of the `DeliteOps` that best describes the operation. For example, since `VectorSum` has the semantics of iterating over the elements of the input Vector and adding them to reduce to a single value, it can be implemented by extending `DeliteOpLoop` with a reduction operation as its body. This significantly reduces the amount of work in implementing a DSL operation since the DSL developers only need to specify the necessary fields of the `DeliteOp` (`range` and `body` in the case of `DeliteOpLoop`) instead of fully implementing the operation.

`DeliteOpLoops` are intended as parallel for-loops. Given an integer index range, the runtime guarantees to execute the loop body exactly once for each index but does not guarantee any execution order. Mutating global state from within a loop is only safe at disjoint indexes. There are specialized constructs to define loop bodies for map and reduce patterns (`DeliteCollectElem`, `DeliteReduceElem`) that transform a collection of elements point-wise or perform aggregation. An optional predicate can be added to perform filter-style operations, i.e. select or aggregate only those elements for which the predicate is true. All loop constructs can be fused into `DeliteOpLoops` that do several operations at once.

Given the relaxed ordering guarantees, the framework can automatically generate efficient parallel code for `DeliteOps`, targeting heterogeneous parallel hardware. Therefore, DSL developers can easily implement parallel DSL operations by extending one of the parallel `DeliteOps`, and only focus on the language design without knowing the low-level details of the target hardware.

The Delite Compiler Framework currently supports Scala, C++, and CUDA targets. The framework provides code generators for each target in addition to a main generator (*Delite generator*) that controls them. The *Delite generator* iterates over the list of available target generators to emit the target-specific kernels. By generating multiple target implementations of the kernels and deferring the decision of which one to use, the framework provides the runtime with enough flexibility in scheduling the kernels based on dynamic information such as resource availability and input size. In addition to the kernels, the Delite generator also generates the *Delite Execution Graph* (DEG) of the application. The DEG is a high-level representation of the program that encodes all necessary information for its execution, including the list of inputs, outputs, and interdependencies of all kernels. After all the kernels are generated, the Delite Runtime starts analyzing the DEG and emits execution plans for each target hardware. Further details are available in Section 16 of Part IV.

# Chapter 12

# Control Abstraction

Among the most useful control abstractions are higher order functions. We can implement support for higher order functions in DSLs while keeping the generated IR strictly first order. This vastly simplifies the compiler implementation and makes optimizations much more effective since the compiler does not have to reason about higher order control flow. We can implement a higher order function `foreach` over Vectors as follows:

```
def infix_foreach[A](v: Rep[Vector[A]])(f: Rep[A] => Rep[Unit]) = {
  var i = 0; while (i < v.length) { f(v(i)); i += 1 }
}
// example:
Vector.rand(100) foreach { i => println(i) }
```

The generated code from the example will be strictly first order, consisting of the unfolded definition of `foreach` with the application of `f` substituted with the `println` statement:

```
while (i < v.length) { println(v(i)); i += 1 }
```

The unfolding is guaranteed by the Scala type system since `f` has type `Rep[A]=>Rep[Unit]`, meaning it will be executed statically but it operates on staged values. In addition to simplifying the compiler, the generated code does not pay any extra overhead. There are no closure allocations and no inlining problems [22].

Other higher order functions like `map` or `filter` could be expressed on top of foreach. Section 11.2.1 and Chapter 16 show how actual Delite DSLs implement these operations as data parallel loops. The rest of this chapter shows how other control structures such as continuations can be supported in the same way.

## 12.1 Leveraging Higher-Order Functions in the Generator

Higher-order functions are extremely useful to structure programs but also pose a significant obstacle for compilers, recent advances on higher-order control-flow analysis notwithstanding [145, 38]. While we would like to retain the structuring aspect for DSL programs, we would like to avoid higher-order control flow in generated code. Fortunately, we can use higher-order functions in the generator stage to compose first-order DSL programs.

Consider the following program that prints the number of elements greater than 7 in some vector:

```
val xs: Rep[Vector[Int]] = ...
println(xs.count(x => x > 7))
```

The program makes essential use of a higher-order function `count` to count the number of elements in a vector that fulfill a predicate given as a function object. Ignoring for the time being that we would likely want to use a `DeliteOp` for parallelism, a good and natural way to implement `count` would be to first define a higher-order function `foreach` to iterate over vectors, as shown at the beginning of the chapter:

```
def infix_foreach[A](v: Rep[Vector[A]])(f: Rep[A] => Rep[Unit]) = {
  var i: Rep[Int] = 0
  while (i < v.length) {
    f(v(i))
    i += 1
  }
}
```

The actual counting can then be implemented in terms of the traversal:

```
def infix_count[A](v: Rep[Vector[A]])(f: Rep[A] => Rep[Boolean]) = {
  var c: Rep[Int] = 0
  v foreach { x => if (f(x)) c += 1 }
  c
}
```

It is important to note that `infix_foreach` and `infix_count` are static methods, i.e. calls will happen at staging time and result in inserting the computed DSL code in the place of the call. Likewise, while the argument vector `v` is a dynamic value, the function argument `f` is again static. However, `f` operates on dynamic values, as made explicit by its type `Rep[A] => Rep[Boolean]`. By contrast, a dynamic function value would have type `Rep[A => B]`.

This means that the code generated for the example program will look roughly like this, assuming that vectors are represented as arrays in the generated code:

```
val v: Array[Int] = ...
var c = 0
var i = 0
while (i < v.length) {
  val x = v(i)
  if (x > 7)
    c += 1
  i += 1
}
println(c)
```

All traces of higher-order control flow have been removed and the program is strictly first-order. Moreover, the programmer can be sure that the DSL program is composed in the desired way.

This issue of higher-order functions is a real problem for regular Scala programs executed on the JVM. The Scala collection library uses essentially the same `foreach` and count `abstractions` as above and the JVM, which applies optimizations based on per-call-site profiling, will identify the call site *within* `foreach` as a hot spot. However, since the number of distinct functions called from foreach is usually large, inlining or other optimizations cannot be applied and every iteration step pays the overhead of a virtual method call [22].

## 12.2 Using Continuations in the Generator to Implement Backtracking

Apart from pure performance improvements, we can use functionality of the generator stage to enrich the functionality of DSLs without any work on the DSL-compiler side. As an example we consider adding backtracking nondeterministic computation to a DSL using a simple variant of McCarthy's `amb` operator [89]. Here is a nondeterministic program that uses `amb` to find pythagorean triples from three given vectors:

```
val u,v,w: Rep[Vector[Int]] = ...
nondet {
  val a = amb(u)
  val b = amb(v)
  val c = amb(w)
  require(a*a + b*b == c*c)
  println("found:")
  println(a,b,c)
}
```

We can use Scala's support for delimited continuations [112] and the associated control operators `shift` and `reset` [33, 32] to implement the necessary primitives. The scope delimiter `nondet` is just the regular `reset`. The other operators are defined as follows:

```
def amb[T](xs: Rep[Vector[T]]): Rep[T] @cps[Rep[Unit]] = shift { k =>
  xs foreach k
}
def require(x: Rep[Boolean]): Rep[Unit] @cps[Rep[Unit]] = shift { k =>
  if (x) k() else ()
}
```

Since the implementation of `amb` just calls the previously defined method `foreach`, the generated code will be first-order and consist of three nested **`while`** loops:

```
val u,v,w:Rep[Vector[Int]]=...
var i = 0
while (i < u.length) {
  val a = u(i)
  val a2 = a*a
  var j = 0
  while (j < v.length) {
    val b = v(j)
    val b2 = b*b
    val a2b2 = a2+b2
    var k = 0
    while (k < w.length) {
      val c = w(k)
      val c2 = c*c
      if (a2b2 == c2) {
        println("found:")
        println(a,b,c)
      }
      k += 1
    }
    j += 1
  }
  i += 1
}
```

Besides the advantage of not having to implement amb as part of the DSL compiler, all common optimizations that apply to plain **while** loops are automatically applied to the unfolded backtracking implementation. For example, the loop invariant hoisting performed by code motion has moved the computation of a*a and b*b out of the innermost loop.

The given implementation of amb is not the only possibility, though. For situations where we know the number of choices (but not necessarily the actual values) for a particular invocation of amb at staging time, we can implement an alternative operator that takes a (static) list of dynamic values and unfolds into specialized code paths for each option at compile time:

```
def bam[T](xs: List[Rep[T]]): Rep[T] @cps[Rep[Unit]] = shift { k =>
  xs foreach k
}
```

Here, foreach is not a DSL operation but a plain traversal of the static argument list xs. The bam operator must be employed with some care because it incurs the risk of code explosion. However, static specialization of nondeterministic code paths can be beneficial if it allows aborting many paths early based on static criteria or merging computation between paths.

```
val u: Rep[Vector[Int]] = ...
nondet {
  val a = amb(u)
  val b = bam(List(x1), List(x2))
  val c = amb(v)
  require(a + c = f(b))  // assume f(b) is expensive to compute
  println("found:")
  println(a,b,c)
}
```

If this example was implemented as three nested loops, f(x1) and f(x2) would need to be computed repeatedly in each iteration of the second loop as they depend on the loop-bound variable b. However, the use of bam will remove the loop over x1,x2 and expose the expensive computations as redundant so that code motion can extract them from the loop:

```
val fx1 = f(x1)
val fx2 = f(x2)
while (...) { // iterate over u
  while (...) { // iterate over v
    if (a + c == fx1) // found
  }
  while (...) { // iterate over v
    if (a + c == fx2) // found
  }
}
```

In principle, the two adjacent inner loops could be subjected to the loop fusion optimization discussed in Section 10.3. This would remove the duplicate traversal of v. In this particular case fusion is currently not applied since it would change the order of the side-effecting println statements.

## 12.3 Using Continuations in the Generator to Generate Async Code Patterns

The previous section used continuations that were completely translated away during generation. In this section, we will use a CPS-transformed program generator to generate code that is in CPS. While the previous section generated only loops, we will generate actual functions in this section, using the mechanisms described in Section 6.2.4. The example is taken from [79] and concerned with generating JavaScript but the techniques apply to any target.[1]

A callback-driven programming style is pervasive in JavaScript programs. Because of lack of thread support, callbacks are used for I/O, scheduling and event-handling. For example, in an Ajax call (Asynchronous JavaScript and XML), one has to provide a callback that will be called once the requested data arrives from the server. This style of programming is known to be unwieldy in more complicated scenarios. To give a specific example, let us consider a scenario where we have an array of Twitter account names and we want to ask Twitter for the latest tweets of each account. Once we obtain the tweets of all users, we would like to log that fact in a console.

We implement this program both directly in JavaScript and in the embedded JavaScript DSL [79]. Let us start by implementing logic that fetches tweets for a single user by using the jQuery library for Ajax calls:

Listing 12.1: Fetching tweets in JavaScript

```
function fetchTweets(user, callback) {
  jQuery.ajax({
    url: "http://api.twitter.com/1/"
     + "statuses/user_timeline.json/",
    type: "GET",
    dataType: "jsonp",
    data: {
      screen_name: user,
      include_rts: true,
      count: 5,
      include_entities: true
    },
    success: callback
  })
}
```

Listing 12.2: Fetching tweets in DSL

```
def fetchTweets(user: Rep[String]) =
  (ajax.get { new JSLiteral {
    val url = "http://api.twitter.com/1/"
          + "statuses/user_timeline.json"
    val `type` = "GET"
    val dataType = "jsonp"
    val data = new JSLiteral {
      val screen_name = user
      val include_rts = true
      val count = 5
      val include_entities = true
    }
  }
}).as[TwitterResponse]
type TwitterResponse =
  Array[JSLiteral {val text: String}]
```

Note that the JavaScript version takes a callback as second argument that will be used to process the fetched tweets. We provide the rest of the logic that iterates over array of users and makes Ajax requests:

[1] *Credits:* Design (mostly) by the author, impl. and presentation by Grzegorz Kossakowski and Nada Amin

Listing 12.3: Twitter example in JavaScript

```
var processed = 0
var users = ["gkossakowski",
  "odersky", "adriaanm"]
users.forEach(function (user) {
  console.log("fetching " + user)
  fetchTweets(user, function(data) {
    console.log("finished " + user)
    data.forEach(function (t) {
      console.log("fetched " + t.text)
    })
    processed += 1
    if (processed == users.length) {
      console.log("done")
    }
  })
})
```

Listing 12.4: Twitter example in DSL

```
val users = array("gkossakowski",
  "odersky", "adriaanm")
for (user <- users.parSuspendable) {
  console.log("fetching " + user)
  val tweets = fetchTweets(user)
  console.log("finished " + user)
  for (t <- tweets)
    console.log("fetched " + t.text)
}
console.log("done")
```

Because of the inverted control flow of callbacks, synchronization between callbacks has to be handled manually. Also, the inverted control flow leads to a code structure that is distant from the programmer's intent. Notice that the in JavaScript version, the call to console that prints “done" is put inside of the foreach loop. If it was put it after the loop, we would get “done” printed *before* any Ajax call has been made leading to counterintuitive behavior.

As an alternative to the callback-driven programming style, one can use an explicit monadic style, possibly sugared by a Haskell-like “do”-notation. However, this requires rewriting possibly large parts of a program into monadic style when a single async operation is added. Another possibility is to automatically transform the program into continuation passing style (CPS), enabling the programmer to express the algorithm in a straightforward, sequential way and creating all the necessary callbacks and book-keeping code automatically. Links [28] uses this approach. However, a whole-program CPS transformation can cause performance degradation, code size blow-up, and stack overflows. In addition, it is not clear how to interact with existing non-CPS libraries as the whole program needs to adhere to the CPS style. Here we use a *selective* CPS transformation, which precisely identifies expressions that need to be CPS transformed.

In fact, the Scala compiler already does selective, `@suspendable` type-annotation-driven CPS transformations of Scala programs [112, 32, 33]. We show how this mechanism can be used for transforming our DSL code before staging and stripping out most CPS abstractions at staging time. The generated JavaScript code does not contain any CPS-specific code but is written in CPS-style by use of JavaScript anonymous functions.

### 12.3.1 CPS and Staging

As an example, we will consider a seemingly blocking `sleep` method implemented in a non-blocking, asynchronous style. In JavaScript, there are no threads and there is no notion of blocking. However the technique is useful in other circumstances as well, for example when

using thread pools, as no thread is being blocked during the sleep period. Let us see how our CPS transformed `sleep` method can be used:

```
def foo() = {
  sleep(1000)
  println("Called foo")
}
reset {
  println("look, Ma ...")
  foo()
  sleep(1000)
  println(" no callbacks!")
}
```

We define `sleep` to use JavaScript's asynchronous `setTimeout` function, which takes an explicit callback:

```
def sleep(delay: Rep[Int]) = shift { k: (Rep[Unit]=>Rep[Unit]) =>
  window.setTimeout(lambda(k), delay)
}
```

The `setTimeout` method expects an argument of type `Rep[Unit=>Unit]` i.e. a staged function of type `Unit=>Unit`. The `shift` method offers us a function of type `Rep[Unit]=>Rep[Unit]`, so we need to reify it to obtain the desired representation. The reification is achieved by the `fun` function (called `lambda` in 6.2.4) provided by our framework and performed at staging time.

It is important to note that reification preserves function composition. Specifically, let `f: Rep[A] => Rep[B]` and `g: Rep[B] => Rep[C]` then `lambda(g compose f) == (lambda(g) compose lambda(f))` where we consider two reified functions to be equal if they yield the same observable effects at runtime. That property of function reification is at the core of reusing the continuation monad in our DSL. Thanks to the fact that the continuation monad composes functions, we can just insert reification at some places (like in a `sleep`) and make sure that we reify *effects* of the continuation monad without the need to reify the monad itself.

### 12.3.2 CPS for Interruptible Traversals

We need to be able to interrupt our execution while traversing an array in order to implement the functionality from listing 12.4. Let us consider a simplified example where we want to sleep during each iteration. We present both JavaScript and DSL code that achieves that (listings 12.5 & 12.6). Both programs, when executed, will print the following output:

```
//pause for 1s
1
//pause for 2s
2
//pause for 3s
3
done
```

Listing 12.5: JavaScript

```
var xs = [1, 2, 3]
var i = 0
var msg = null
function f1() {
  if (i < xs.length) {
    window.setTimeout(f2, xs[i]*1000)
    msg = xs[i]
    i++
  }
}
function f2() {
  console.log(msg)
  f1()
}
f1()
```

Listing 12.6: DSL

```
val xs = array(1, 2, 3)
// shorthand for xs.suspendable.foreach
for (x <- xs.suspendable) {
  sleep(x * 1000)
  console.log(String.valueOf(x))
}
log("done")
```

In the DSL code, we use a `suspendable` variant of arrays, which is achieved through an implicit conversion from regular arrays:

```
implicit def richArray(xs: Rep[Array[A]]) = new {
  def suspendable: SArrayOps[A] = new SArrayOps[A](xs)
}
```

The idea behind `suspendable` arrays is that iteration over them can be interrupted. We will have a closer look at how to achieve that with the help of CPS. The `suspendable` method returns a new instance of the `SArrayOps` class defined here:

Listing 12.7: Suspendable foreach

```
class SArrayOps(xs: Rep[Array[A]]) {
  def foreach(yld: Rep[A] => Rep[Unit] @suspendable):
    Rep[Unit] @suspendable = {
      var i = 0
      suspendableWhile(i < xs.length) { yld(xs(i)); i += 1 }
    }
}
```

Note that one cannot use while loops in CPS but we can simulate them with recursive functions. Let us see how regular while loop can be simulated with a recursive function:

```
def recursiveWhile(cond: => Boolean)(body: => Unit): Unit = {
  def rec = () => if (cond) { body; rec() } else {}
  rec()
}
```

By adding CPS-related declarations and control abstractions, we implement `suspendableWhile`:

```
def suspendableWhile(cond: => Rep[Boolean])(
  body: => Rep[Unit] @suspendable): Rep[Unit] @suspendable =
  shift { k =>
```

```
    def rec = fun { () =>
      if (cond) reset { body; rec() } else { k() }
    }
    rec()
  }
```

### 12.3.3 Defining the Ajax API

With the abstractions for interruptible loops and traversals at hand, what remains to complete the Twitter example from the beginning of the section is the actual Ajax request/response cycle.

The Ajax interface component provides a type `Request` that captures the request structure expected by the underlying JavaScript/jQuery implementation and the necessary object and method definitions to enable the use of `ajax.get` in user code:

```
trait Ajax extends JS with CPS {
  type Request = JSLiteral {
    val url: String
    val `type`: String
    val dataType: String
    val data: JSLiteral
  }
  type Response = Any
  object ajax {
    def get(request: Rep[Request]) = ajax_get(request)
  }
  def ajax_get(request: Rep[Request]): Rep[Response] @suspendable
}
```

Notice that the `Request` type is flexible enough to support an arbitrary object literal type for the `data` field through subtyping. The `Response` type alias points at `Any` which means that it is the user's responsibility to either use `dynamic` or perform an explicit cast to the expected data type.

The corresponding implementation component implements `ajax_get` to capture a continuation, reify it as a staged function using `fun` and store it in an `AjaxGet` IR node.

```
trait AjaxExp extends JSExp with Ajax {
  case class AjaxGet(request: Rep[Request],
    success: Rep[Response => Unit]) extends Def[Unit]
  def ajax_get(request: Rep[Request]): Rep[Response] @suspendable =
    shift { k =>
      reflectEffect(AjaxGet(request, lambda(k)))
    }
}
```

During code generation, we emit code to attach the captured continuation as a callback function in the `success` field of the request object:

```
trait GenAjax extends JSGenBase {
  val IR: AjaxExp
  import IR._
  override def emitNode(sym: Sym[Any], rhs: Def[Any])(
    implicit stream: PrintWriter) = rhs match {
      case AjaxGet(req, succ) =>
        stream.println(quote(req) + ".success = " + quote(succ))
        emitValDef(sym, "jQuery.ajax(" + quote(req) + ")")
    case _ => super.emitNode(sym, rhs)
  }
}
```

It is interesting to note that, since the request already has the right structure for the `jQuery.ajax` function, we can simply pass it to the framework-provided `quote` method, which knows how to generate JavaScript representations of any `JSLiteral`.

The Ajax component completes the functionality required to run the Twitter example with one caveat: The outer loop in listing 12.4 uses `parSuspendable` to traverse arrays instead of the `suspendable` traversal variant we have defined in listing 12.7.

In fact, if we change the code to use `suspendable` instead of `parSuspendable` and run the generated JavaScript program, we will get following output printed to the JavaScript console:

```
fetching gkossakowski
finished fetching gkossakowski
fetched [...]
fetched [...]
fetching odersky
finished fetching odersky
fetched [...]
fetched [...]
fetching adriaanm
finished fetching adriaanm
fetched [...]
fetched [...]
done
```

Notice that all Ajax requests were done sequentially. Specifically, there was just one Ajax request active at a given time; when the callback to process one request is called, it would resume the continuation to start another request, and so on. In many cases this is exactly the desired behavior, however, we will most likely want to perform our Ajax request in parallel.

### 12.3.4 CPS for Parallelism

The goal of this section is to implement a parallel variant of the `foreach` method from listing 12.7. We will start with defining a few primitives like futures and dataflow cells. We start with cells, which we decide to define in JavaScript, as another example of integrating external libraries with our DSL:

Listing 12.8: JavaScript-based implementation of a non-blocking Cell

```
function Cell() {
  this.value = undefined
  this.isDefined = false
  this.queue = []
  this.get = function (k) {
    if (this.isDefined) {
      k(this.value)
    } else {
      this.queue.push(k)
    }
  }
  this.set = function (v) {
    if (this.isDefined) {
      throw "can't set value twice"
    } else {
      this.value = v
      this.isDefined = true
      this.queue.forEach(function (f) {
        f(v) //non-trivial spawn could be used here
      })
    }
  }
}
```

A cell object allows us to track dependencies between values. Whenever the `get` method is called and the value is not in the cell yet, the continuation will be added to a queue so it can be suspended until the value arrives. The `set` method takes care of resuming queued continuations. We expose `Cell` as external library using our typed API mechanism and we use it for implementing an abstraction for futures.

```
def createCell(): Rep[Cell[A]]
trait Cell[A]
trait CellOps[A] {
  def get(k: Rep[A => Unit]): Rep[Unit]
  def set(v: Rep[A]): Rep[Unit]
}
implicit def repToCellOps(x: Rep[Cell[A]]): CellOps[A] =
  repProxy[Cell[A],CellOps[A]](x)

def spawn(body: => Rep[Unit] @suspendable): Rep[Unit] = {
  reset(body) //non-trivial implementation uses
              //trampolining to prevent stack overflows
}
def future(body: => Rep[A] @suspendable) = {
  val cell = createCell[A]()
  spawn { cell.set(body) }
  cell
```

```
}
```

The last bit of general functionality we need is `RichCellOps` that ties `Cell`s and continuations together inside of our DSL.

```
class RichCellOps(cell: Rep[Cell[A]]) {
  def apply() = shift { k: (Rep[A] => Rep[Unit]) =>
    cell.get(lambda(k))
  }
}
implicit def richCellOps(x: Rep[Cell[A]]): RichCell[A] =
  new RichCellOps(x)
```

It is worth noting that `RichCellOps` is not reified so it will be dropped at staging time and its method will get inlined whenever used. Also, it contains CPS-specific code that allows us to capture the continuation. The `fun` function reifies the captured continuation.

We are ready to present the parallel version of `foreach` defined in listing 12.7.

```
def foreach(yld: Rep[A] => Rep[Unit] @suspendable):
  Rep[Unit] @suspendable = {
    val futures = xs.map(x => future(yld(x)))
    futures.suspendable.foreach(_.apply())
  }
```

We instantiate each future separately so they can be executed in parallel. As a second step we make sure that all futures are evaluated before we leave the `foreach` method by forcing evaluation of each future and "waiting" for its completion. Thanks to CPS transformations, all of that will be implemented in a non-blocking style.

The only difference between the parallel and serial versions of the Twitter example 12.4 is the use of `parSuspendable` instead of `suspendable` so the parallel implementation of the `foreach` method is used. The rest of the code stays the same. It is easy to switch between both versions, and users are free to make their choice according to their needs and performance requirements.

## 12.4 Guarantees by Construction

Making staged functions explicit through the use of `lambda` (as described in Section 6.2.4) enables tight control over how functions are structured and composed. For example, functions with multiple parameters can be specialized for a subset of the parameters. Consider the following implementation of Ackermann's function:

```
  def ack(m: Int): Rep[Int=>Int] = lambda { n =>
    if (m == 0) n+1 else
    if (n == 0) ack(m-1)(1) else
    ack(m-1)(ack(m)(n-1))
  }
```

Calling `ack(m)(n)` will produce a set of mutually recursive functions, each specialized to a particular value of `m` (example `m=2`):

```
def ack_2(n: Int) = if (n == 0) ack_1(1) else ack_1(ack_2(n-1))
def ack_1(n: Int) = if (n == 0) ack_0(1) else ack_0(ack_1(n-1))
def ack_0(n: Int) = n+1
acc_2(n)
```

In essence, this pattern implements what is known as "polyvariant specialization" in the partial evaluation community. But unlike automatic partial evaluation, which might or might not be able to discover the *right* specialization, the use of staging provides a strong guarantee about the structure of the generated code.

Other strong guarantees can be achieved by restricting the interface of function definitions. Being of type `Rep[A=>B]`, the result of `lambda` is a first-class value in the generated code that can be stored or passed around in arbitrary ways. However we might want to avoid higher-order control flow in generated code for efficiency reasons, or to simplify subsequent analysis passes. In this case, we can define a new function constructor `fundef` as follows:

```
def fundef[A,B](f: Rep[A] => Rep[B]): Rep[A] => Rep[B] =
  (x: Rep[A]) => lambda(f).apply(x)
```

Using `fundef` instead of `lambda` produces a restricted function that can only be applied but not passed around in the generated code (type `Rep[A]=>Rep[B]`). At the same time, a result of `fundef` is still a first class value in the code *generator*. If we do not expose `lambda` and `apply` at all to client code, we obtain a guarantee that each function call site unambiguously identifies the function definition being called and no closure objects will need to be created at runtime.

# Chapter 13

# Data Abstraction

High level data structures are a cornerstone of modern programming and at the same time stand in the way of compiler optimizations.

As a running example we consider implementing a complex number datatype in a DSL. The usual approach of languages executed on the JVM is to represent every non-primitive value as a heap-allocated reference object. The space overhead, reference indirection as well as the allocation and garbage collection cost are a burden for performance critical code. Thus, we want to be sure that our complex numbers can be manipulated as efficiently as two individual doubles. In the following, we explore different ways to achieve that.

## 13.1 Static Data Structures

The simplest approach is to implement complex numbers as a fully static data type, that only exists at staging time. Only the actual `Doubles` that constitute the real and imaginary components of a complex number are dynamic values:

```
case class Complex(re: Rep[Double], im: Rep[Double])
def infix_+(a: Complex, b: Complex) =
  Complex(a.re + b.re, a.im + b.im)
def infix_*(a: Complex, b: Complex) =
  Complex(a.re*b.re - a.im*b.im, a.re*b.im + a.im*b.re)
```

Given two complex numbers `c1,c2`, an expression like

```
c1 + 5 * c2  // assume implicit conversion from Int to Complex
```

will generate code that is free of `Complex` objects and only contains arithmetic on `Doubles`.

However the ways we can use `Complex` objects are rather limited. Since they only exists at staging time we cannot, for example, express dependencies on dynamic conditions:

```
val test: Rep[Boolean] = ...
val c3 = if (test) c1 else c2 // type error: c1/c2 not a Rep type
```

It is worthwhile to point out that nonetheless, purely static data structures have important use cases. To give an example, the fast fourier transform (FFT) [27] is branch-free for a fixed

input size. The definition of complex numbers given above can be used to implement a staged FFT that computes the well-known butterfly shaped computation circuits from the textbook Cooley-Tukey recurrences (see Section 14.3).

To make complex numbers work across conditionals, we have have to split the control flow explicitly (another option would be using mutable variables). There are multiple ways to achieve this splitting. We can either duplicate the test and create a single result object:

```
val test: Rep[Boolean] = ...
val c3 = Complex(if (test) c1.re else c2.re, if (test) c1.im else c2.im)
```

Alternatively we can use a single test and duplicate the rest of the program:

```
val test: Rep[Boolean] = ...
if (test) {
  val c3 = c1
  // rest of program
} else {
  val c3 = c2
  // rest of program
}
```

While it is awkward to apply this transformation manually, we can use continuations (much like for the `bam` operator in Section 12.2) to generate two specialized computation paths:

```
def split[A](c: Rep[Boolean]) = shift { k: (Boolean => A) =>
  if (c) k(true) else k(false) // "The Trick"
}
val test: Rep[Boolean] = ...
val c3 = if (split(test)) c1 else c2
```

The generated code will be identical to the manually duplicated, specialized version above.

## 13.2 Dynamic Data Structures with Partial Evaluation

We observe that we can increase the amount of statically possible computation (in a sense, applying binding-time improvements) for dynamic values with domain-specific rewritings:

```
val s: Int = ...            // static
val d: Rep[Int] = ...       // dynamic

val x1 = s + s + d          // left assoc: s + s evaluated statically,
                            // one dynamic addition
val x2 = s + (d + s)        // naively: two dynamic additions,
                            // using pattern rewrite: only one
```

In computing `x1`, there is only one dynamic addition because the left associativity of the plus operator implies that the two static values will be added together at staging time. Computing `x2` will incur two dynamic additions, because both additions have at least one dynamic summand. However we can add rewriting rules that first replace `d+c` (`c` denoting a

dynamic value that is know to be a static constant, i.e. an IR node of type `Const`) with `c+d` and then `c+(c+d)` with `(c+c)+d`. The computation `c+c` can again be performed statically.

We have seen in Section 10.2.2 how we can define a generic framework for data structures that follows a similar spirit. The interface for field accesses `field` pattern matches on its argument and, if that is a `Struct` creation, looks up the desired value from the embedded hash map.

An implementation of complex numbers in terms of `Struct` could look like this:

```
trait ComplexOps extends ComplexBase with ArithOps {
  def infix_+(x: Rep[Complex], y: Rep[Complex]): Rep[Complex] =
    Complex(x.re + y.re, x.im + y.im)
  def infix_*(x: Rep[Complex], y: Rep[Complex]): Rep[Complex] =
    Complex(a.re*b.re - ...)
}
trait ComplexBase extends Base {
  class Complex
  def Complex(re: Rep[Double], im: Rep[Double]): Rep[Complex]
  def infix_re(c: Rep[Complex]): Rep[Double]
  def infix_im(c: Rep[Complex]): Rep[Double]
}
trait ComplexStructExp extends ComplexBase with StructExp {
  def Complex(re: Rep[Double], im: Rep[Double]) =
    struct[Complex](classTag("Complex"), Map("re"->re, "im"->im))
  def infix_re(c: Rep[Complex]): Rep[Double] = field[Double](c, "re")
  def infix_im(c: Rep[Complex]): Rep[Double] = field[Double](c, "im")
}
```

Note how complex arithmetic is defined completely within the interface trait `ComplexOps`, which inherits double arithmetic from `ArithOps`. Access to the components via `re` and `im` is implemented using `struct`.

Using virtualized record types (see Section 5.1.3) that map to `struct` internally, we can express the type definition more conveniently as

```
class Complex extends Struct { val re: Double, val im: Double }
```

and remove the need for methods `infix_re` and `infix_im`. The Scala-Virtualized compiler will automatically provide staged field accesses like `c.re` and `c.im`. It is still useful to add a simplified constructor method

```
def Complex(r: Rep[Double], i: Rep[Double]) =
  new Complex { val re = re; val im = im }
```

to enable using `Complex(re,im)` instead of the `new Complex` syntax.

In contrast to the completely static implementation of complex numbers presented in Section 13.1 above, complex numbers are a fully dynamic DSL type now. The previous restrictions are gone and we can write the following code without compiler error:

```
val c3 = if (test) c1 else c2
println(c3.re)
```

The conditional `ifThenElse` is overridden to split itself for each field of a struct. Internally the above will be represented as:

```
val c3re = if (test) c1re else c2re
val c3im = if (test) c1im else c2im   // removed by dce
val c3 = Complex(c3re, c3im)          // removed by dce
println(c3re)
```

The computation of the imaginary component as well as the struct creation for the result of the conditional are never used and thus they will be removed by dead code elimination.

## 13.3 Generic Programming with Type Classes

The type class pattern [154], which decouples data objects from generic dispatch, fits naturally with a staged programming model as type class instances can be implemented as static objects.

Extending the Vector example, we might want to be able to add vectors that contain numeric values. We can use a lifted variant of the `Numeric` type class from the Scala library

```
class Numeric[T] {
  def num_plus(a: Rep[T], b: Rep[T]): Rep[T]
}
```

and provide a type class instance for complex numbers:

```
implicit def complexIsNumeric = new Numeric[Complex] {
  def num_plus(a: Rep[Complex], b: Rep[Complex]) = a + b
}
```

Generic addition on Vectors is straightforward, assuming we have a method `zipWith` already defined:

```
def infix_+[T:Numeric](a: Rep[Vector[T]], b: Rep[Vector[T]]) = {
  val m = implicitly[Numeric[T]] // access type class instance
  a.zipWith(b)((u,v) => m.num_plus(u,v))
}
```

With that definition at hand we can add a type class instance for numeric vectors:

```
implicit def vecIsNumeric[T:Numeric] = new Numeric[Vector[T]] {
  def num_plus(a: Rep[Vector[T]], b: Rep[Vector[T]]) = a + b
```

which allows us to pass, say, a `Rep[Vector[Complex]]` to any function that works over generic types `T:Numeric` including vector addition itself. The same holds for nested vectors of type `Rep[Vector[Vector[Complex]]]`. Usually, type classes are implemented by passing an implicit dictionary, the type class instance, to generic functions. Here, type classes are a purely stage-time concept. All generic code is specialized to the concrete types and no type class instances exist (and hence no virtual dispatch occurs) when the DSL program is run.

An interesting extension of the type class model is the notion of polytypic staging, studied on top of LMS [119].

## 13.4 Unions and Inheritance

The struct abstraction from Section 10.2.2 can be extended to sum types and inheritance using a tagged union approach [98, 66]. We add a `clzz` field to each struct that refers to an expression that defines the object's class. Being a regular struct field, it is subject to all common optimizations. We extend the complex number example with two subclasses:

```
abstract class Complex
class Cartesian extends Complex with Struct { val re: Double, val im: Double }
class Polar extends Complex with Struct { val r: Double, val phi: Double }
```

Splitting transforms work as before: e.g. conditional expressions are forwarded to the fields of the struct. But now the result struct will contain the union of the fields found in the two branches, inserting null values as appropriate. A conditional is created for the `clzz` field only if the exact class is not known at staging time. As an example, the expression

```
val a = Cartesian(1.0, 2.0); val b = Polar(3.0, 4.0)
if (x > 0) a else b
```

produces this generated code:

```
val (re, im, r, phi, clzz) =
  if (x > 0) (1.0, 2.0, null, null, classOf[Cartesian])
  else (null, null, 3.0, 4.0, classOf[Polar])
struct("re"->re, "im"->im, "r"->r, "phi"->phi, "clzz"->clzz)
```

The `clzz` fields allows virtual dispatch via type tests and type casting, e.g. to convert any complex number to its cartesian representation:

```
def infix_toCartesian(c: Rep[Complex]): Rep[Cartesian] =
  if (c.isInstanceOf[Cartesian]) c.asInstanceOf[Cartesian]
  else { val p = c.asInstanceOf[Polar]
    Cartesian(p.r * cos(p.phi), p.r * sin(p.phi)) }
```

Appropriate rewrites ensure that if the argument is known to be a Cartesian, the conversion is a no-op. The type test that inspects the clzz field is only generated if the type cannot be determined statically. If the clzz field is never used it will be removed by DCE.

## 13.5 Struct of Array and Other Data Format Conversions

There is another particularly interesting use case for the splitting of data structures: Let us assume we want to create a vector of complex numbers. Just as with the if-then-else example above, we can override the vector constructors such that a `Vector[Cartesian]` is represented as a struct that contains two separate arrays, one for the real and one for the imaginary components. A more general `Vector[Complex]` that contains both polar and cartesian values will be represented as five arrays, one for each possible data field plus the `clzz` tag for each value. In fact, we have expressed our conceptual array of structs as a struct of arrays (AoS to SoA transform, see Section 10.2.3). This data layout is beneficial in many cases. Consider for

example calculating complex conjugates (i.e. swapping the sign of the imaginary components) over a vector of complex numbers.

```
def conj(c: Rep[Complex]) = if (c.isCartesian) {
  val c2 = c.toCartesian; Cartesian(c2.re, -c2.im)
} else {
  val c2 = c.toPolar; Polar(c2.r, -c2.phi)
}
```

To make the test case more interesting we perform the calculation only in one branch of a conditional.

```
val vector1 = ... // only Cartesian values
if (test) {
  vector1.map(conj)
} else {
  vector1
}
```

All the real parts remain unchanged so the array holding them need not be touched at all. Only the imaginary parts have to be transformed, cutting the total required memory bandwidth in half. Uniform array operations like this are also a much better fit for SIMD execution. The generated intermediate code is:

```
val vector1re = ...
val vector1im = ...
val vector1clzz = ... // array holding classOf[Cartesian] values
val vector2im = if (test) {
  Array.fill(vector1size) { i => -vector1im(i) }
} else {
  vector1im
}
struct(ArraySoaTag(Complex,vector1size),
  Map("re"->vector1re, "im"->vector2im, "clzz"->vector1clzz))
```

Note how the conditionals for the `re` and `clzz` fields have been eliminated since the fields do not change (the initial array contained cartesian numbers only). If the struct expression will not be referenced in the final code, dead code elimination removes the `clzz` array.

In the presence of conditionals that produce array elements of different types, it can be beneficial to use a sparse representation for arrays that make up the result struct-of-array, similar to the approach in Data Parallel Haskell [66]. Of course no choice of layout is optimal in all cases, so the usual sparse versus dense tradeoffs regarding memory use and access time apply here as well.

We conclude this section by taking note that we can actually guarantee that no dynamic `Complex` or `Struct` object is ever created just by not implementing code generation logic for `Struct` and `Field` IR nodes and signaling an error instead. This is a good example of a performance-oriented DSL compiler rejecting a program as ill-formed because it cannot be executed in the desired, efficient way.

## 13.6 Loop Fusion and Deforestation

Building complex bulk operations out of simple ones often leads to inefficient generated code. For example consider the simple vector code

```
val a: Rep[Double] = ...
val x: Rep[Vector[Double]] = ...
val y: Rep[Vector[Double]] = ...

a*x+y
```

Assuming we have provided the straightforward loop-based implementations of scalar-times-vector and vector-plus-vector, the resulting code for this program will perform two loops and allocate a temporary vector to store `a*x`. A more efficient implementation will only use a single loop (and no temporary vector allocations) to compute `a*x(i)+y(i)`.

In addition to operations that are directly dependent as illustrated above, side-by-side operations also appear frequently. As an example, consider a DSL that provides mean and variance methods.

```
def mean(x: Rep[Vector[Double]]) =
    sum(x.length) { i => x(i) } / x.length
def variance(x: Rep[Vector[Double]]) =
    sum(x.length) { i => square(x(i)) } / x.length - square(mean(x))

val data = ...

val m = mean(data)
val v = variance(data)
```

The DSL developer wishes to provide these two functions separately, but many applications will compute both the mean and variance of a dataset together. In this case we again want to perform all the work with a single pass over `data`. In both of the above example situations, fusing the operations into a single loop greatly improves cache behavior and reduces the total number of loads and stores required. It also creates coarser-grained functions out of fine-grained ones, which will likely improve parallel scalability.

Our framework handles all situations like these two examples uniformly and for all DSLs. Any non-effectful loop IR node is eligible for fusing with other loops. In order to handle all the interesting loop fusion cases, the fusing algorithm uses simple and general criteria: It fuses all pairs of loops where either both loops have the exact same size or one loop iterates over a data structure the other loop creates, as long as fusing will not create any cyclic dependencies. The exact rules are presented in Section 10.3. When it finds two eligible loops the algorithm creates a new loop with a body composed of both of the original bodies. Merging loop bodies includes array contraction, i.e. the fusing transform modifies dependencies so that all results produced within a loop iteration are consumed directly rather than by reading an output data structure. Whenever this renders an output data structure unnecessary (it does not escape the fused loop) it is removed automatically by dead code elimination. All DeliteOpLoops are parallel loops, which allows the fused loops to be parallelized in the same manner as the original loops.

The general heuristic is to apply fusion greedily wherever possible. For dominantly imperative code more refined heuristics might be needed [8]. However, our loop abstractions are dominantly functional and many loops create new data structures. Removing intermediate data buffers, which are potentially large and many of which are used only once is clearly a win, so fusing seems to be beneficial in almost all cases.

Our fusion mechanism is similar but not identical to deforestation [153] and related approaches [30]. Many of these approaches only consider expressions that are directly dependent (vertical fusion), whereas we are able to handle both dependent and side-by-side expressions (horizontal fusion) with one general mechanism. This is critical for situations such as the mean and variance example, where the only other efficient alternative would be to explicitly create a composite function that returns both results simultaneously. This solution additionally requires the application writer to always remember to use the composite version when appropriate. It is generally difficult to predict all likely operation compositions as well as onerous to provide efficient, specialized implementations of them. Therefore fusion is key for efficient compositionality in both applications and DSL libraries.

## 13.7 Extending the Framework

A framework for building DSLs must be easily extensible in order for the DSL developer to exploit domain knowledge starting from a general-purpose IR design. Consider a simple DSL for linear algebra with a Vector type. Now we want to add norm and dist functions to the DSL. The first possible implementation is to simply implement the functions as library methods.

```
def norm[T:Numeric](v: Rep[Vector[T]]) = {
  sqrt(v.map(j => j*j).sum)
}
def dist[T:Numeric](v1: Rep[Vector[T]], v2: Rep[Vector[T]]) = {
  norm(v1 - v2)
}
```

Whenever the dist method is called the implementation will be added to the application IR in terms of vector subtraction, vector map, vector sum, etc. (assuming each of these methods is built-in to the language rather than also being provided as a library method). This version is very straightforward to write but the knowledge that the application wishes to find the distance between two vectors is lost.

By defining norm explicitly in the IR implementation trait (where Rep[T] = Exp[T]) we gain ability to perform pattern matching on the IR nodes that compose the arguments.

```
override def norm[T:Numeric](v: Exp[Vector[T]]) = v match {
  case Def(ScalarTimesVector(s,u)) => s * norm(u)
  case Def(ZeroVector(n)) => 0
  case _ => super.norm(v)
}
```

In this example there are now three possible implementations of `norm`. The first case factors scalar-vector multiplications out of `norm` operations, the second short circuits the norm of a

ZeroVector to be simply the constant 0, and the third falls back on the default implementation defined above. With this method we can have a different implementation of `norm` for each *occurrence* in the application.

An even more powerful alternative is to implement `norm` and `dist` as custom IR nodes. This enables the DSL to include these nodes when optimizing the application via pattern matching and IR rewrites as illustrated above. For example, we can add a rewrite rule for calculating the norm of a unit vector: if $v_1 = \frac{v}{\|v\|}$ then $\|v_1\| = 1$. In order to implement this optimization we need to add cases both for the new `norm` operation as well as to the existing scalar-times-vector operation to detect the first half of the pattern.

```
case class VectorNorm[T](v: Exp[Vector[T]]) extends Def[T]
case class UnitVector[T](v: Exp[Vector[T]]) extends Def[Vector[T]]

override def scalar_times_vector[T:Numeric](s: Exp[T], v: Exp[Vector[T]]) =
(s,v) match {
  case (Def(Divide(Const(1), Def(VectorNorm(v1)))), v2)
    if v1 == v2 => UnitVector(v)
  case _ => super.scalar_times_vector(s,v)
}
override def norm[T:Numeric](v: Exp[Vector[T]]) = v match {
  case Def(UnitVector(v1)) => 1
  case _ => super.norm(v)
}
```

In this example the scalar-times-vector optimization requires vector-norm to exist as an IR node to detect[1] and short-circuit the operation to simply create and mark unit vectors. The vector-norm optimization then detects unit vectors and short circuits the norm operation to simply add the constant 1 to the IR. In every other case it falls back on the default implementation, which is to create a new `VectorNorm` IR node.

The default constructor for `VectorNorm` uses delayed rewriting (see Section 10.1.3) to specify the desired lowering of the IR node:

```
def norm[T:Numeric](v: Rep[Vector[T]]) = VectorNorm(v) atPhase(lowering) {
  sqrt(v.map(j => j*j).sum)
}
```

The right hand side of this translation is exactly the initial norm implementation we started with.

[1] The == operator tests structural equality of IR nodes. The test is cheap because we only need to look at symbols, one level deep. Value numbering/CSE ensures that intensionally equal IR nodes get assigned the same symbol.

# Chapter 14

# Case Studies

This chapter presents case studies for Delite apps (using the OptiML and OptiQL DSLs) as well as classical staging use cases (FFT specialization and regular expression matching). The Delite apps are real-world examples for the loop fusion algorithm from Section 10.3 and the struct conversion from Section 10.2.2.

## 14.1 OptiML Stream Example

[1] OptiML is an embedded DSL for machine learning (ML) developed on top of LMS and Delite. It provides a MATLAB-like programming model with ML-specific abstractions. OptiML is a prototypical example of how the techniques described in this thesis can be used to construct productive, high performance DSLs targeted at heterogeneous parallel machines.

### 14.1.1 Downsampling in Bioinformatics

In this example, we will demonstrate how the optimization and code generation techniques discussed in previous sections come together to produce efficient code in real applications. SPADE is a bioinformatics application that builds tree representations of large, high-dimensional flow cytometry datasets. Consider the following small but compute-intensive snippet from SPADE (C++):

```
std::fill(densities, densities+obs, 0);
#pragma omp parallel for shared(densities)
for (size_t i=0; i<obs; i++) {
  if (densities[i] > 0)
    continue;
  std::vector<size_t> apprxs;  // Keep track on observations we can approximate
  Data_t *point = &data[i*dim];
  Count_t c = 0;

  for (size_t j=0; j<obs; j++) {
```

[1] *Credits:* Design and presentation by Arvind Sujeeth, fusion implementation by the author

```
    Dist_t d = distance(point, &data[j*dim], dim);
    if (d < apprx_width) {
      apprxs.push_back(j);
      c++;
    } else if (d < kernel_width) c++;
  }
  // Potential race condition on other density entries, use atomic
  // update to be safe
  for (size_t j=0; j<apprxs.size(); j++)
    __sync_bool_compare_and_swap(densities+apprxs[j],0,c);
  densities[i] = c;
}
```

This snippet represents a downsampling step that computes a set of values, densities, that represents the number of samples within a bounded distance (kernel_width) from the current sample. Furthermore, any distances within apprx_width of the current sample are considered to be equivalent, and the density for the approximate group is updated as a whole. Finally, the loop is run in parallel using OpenMP. This snippet represents hand-optimized, high performance, low-level code. It took a systems and C++ expert to port the original MATLAB code (written by a bioinformatics researcher) to this particular implementation. In contrast, consider the equivalent snippet of code, but written in OptiML:

```
val distances = Stream[Double](data.numRows, data.numRows) {
  (i,j) => dist(data(i),data(j))
}
val densities = Vector[Int](data.numRows, true)

for (row <- distances.rows) {
  if(densities(row.index) == 0) {
    val neighbors = row find { _ < apprxWidth }
    densities(neighbors) = row count { _ < kernelWidth }
  }
}
densities
```

This snippet is expressive and easy to write. It is not obviously high performance. However, because we have abstracted away implementation detail, and built-in high-level semantic knowledge into the OptiML compiler, we can generate code that is essentially the same as the hand-tuned C++ snippet. Let's consider the OptiML code step by step.

Line 1 instantiates a Stream, which is an OptiML data structure that is buffered; it holds only a chunk of the backing data in memory at a time, and evaluates operations one chunk at a time. Stream only supports iterator-style access and bulk operations. These semantics are necessary to be able to express the original problem in a more natural way without adding overwhelming performance overhead. The foreach implementation for stream.rows is:

```
def stream_foreachrow[A:Manifest](x: Exp[Stream[A]],
              block: Exp[StreamRow[A]] => Exp[Unit]) = {
  var i = 0
```

```
  while (i < numChunks) {
    val rowsToProcess = stream_rowsin(x, i)
    val in = (0::rowsToProcess)
    val v = fresh[Int]

    // fuse parallel initialization and foreach function
    reflectEffect(StreamInitAndForeachRow(in, v, x, i, block))   // parallel
    i += 1
  }
}
```

This method constructs the IR nodes for iterating over all of the chunks in the Stream, initalizing each row, and evaluating the user-supplied foreach anonymous function. We first obtain the number of rows in the current chunk by calling a method on the Stream instance (`stream_rowsin`). We then call the StreamInitAndForeachRow op, which is a DeliteOpForeach, over all of the rows in the chunk. OptiML unfolds the foreach function and the stream initialization function while building the IR, inside StreamInitAndForeachRow. The stream initialization function (`((i,j) => dist(data(i),data(j))`) constructs a StreamRow, which is the input to the foreach function. The representation of the foreach function consists of an IfThenElse operation, where the then branch contains the VectorFind, VectorCount, and VectorBulkUpdate operations from lines 6-7 of the OptiML SPADE snippet. VectorFind and VectorCount both extend DeliteOpLoop. Since they are both DeliteOpLoops over the same range with no cyclic dependencies, they are fused into a single DeliteOpLoop. This eliminates an entire pass (and the corresponding additional memory accesses) over the row, which is a non-trivial 235,000 elements in one typical dataset.

Fusion helps to transform the generated code into the iterative structure of the C++ code. One important difference remains: we only want to compute the distance if it hasn't already been computed for a neighbor. In the streaming version, this corresponds to only evaluating a row of the Stream if the user-supplied if-condition is true. In other words, we need to optimize the initialization function *together with* the anonymous function supplied to the foreach. LMS does this naturally since the foreach implementation and the user code written in the DSL are all uniformly represented with the same IR. When the foreach block is scheduled, the stream initialization function is pushed inside the user conditional because the StreamRow result is not required anywhere else. Furthermore, once the initialization function is pushed inside the conditional, it is then fused with the existing DeliteOpLoop, eliminating another pass. We can go even further and remove all dependencies on the StreamRow instance by bypassing field accesses on the row, using the pattern matching mechanism described earlier:

```
trait StreamOpsExpOpt extends StreamOpsExp {
  this: OptiMLExp with StreamImplOps =>

  override def stream_numrows[A:Manifest](x: Exp[Stream[A]]) = x match {
    case Def(Reflect(StreamObjectNew(numRows, numCols,
                     chunkSize, func, isPure),_,_)) => numRows
    case _ => super.stream_numrows(x)
```

```
  }
  // similar overrides for other stream fields
}
trait VectorOpsExpOpt extends VectorOpsExp {
  this: OptiMLExp with VectorImplOps =>
  // accessing an element of a StreamRow directly accesses the underlying Stream
  override def vector_apply[A:Manifest](x: Exp[Vector[A]], n: Exp[Int]) = x match {
    case Def(StreamChunkRow(x, i, offset)) => stream_chunk_elem(x,i,n)
    case _ => super.vector_apply(x,n)
  }
}
```

Now as the row is computed, the results of VectorFind and VectorCount are also computed in a pipelined fashion. All accesses to the StreamRow are short-circuited to their underlying data structure (the Stream), and no StreamRow object is ever allocated in the generated code. The following listing shows the final code generated by OptiML for the "then" branch (comments and indentation added for clarity):

```
// ... initialization code omitted ...
// -- FOR EACH ELEMENT IN ROW --
while (x155 < x61) {
  val x168 = x155 * x64
  var x185: Double = 0
  var x180 = 0

  // -- INIT STREAM VALUE (dist(i,j))
  while (x180 < x64) {
    val x248 = x164 + x180
    val x249 = x55(x248)
    val x251 = x168 + x180
    val x252 = x55(x251)
    val x254 = x249 - x252
    val x255 = java.lang.Math.abs(x254)
    val x184 = x185 + x255
    x185 = x184
    x180 += 1
  }
  val x186 = x185
  val x245 = x186 < 6.689027961000001
  val x246 = x186 < 22.296759870000002

  // -- VECTOR FIND --
  if (x245) x201.insert(x201.length, x155)

  // -- VECTOR COUNT --
  if (x246) {
    val x207 = x208 + 1
    x208 = x207
  }
  x155 += 1
}

// -- VECTOR BULK UPDATE --
var forIdx = 0
while (forIdx < x201.size) {
  val x210 = x201(forIdx)
  val x211 = x133(x210) = x208
  x211
  forIdx += 1
}
```

This code, though somewhat obscured by the compiler generated names, closely resembles the hand-written C++ snippet shown earlier. It was generated from a simple, 9 line description of the algorithm written in OptiML, making heavy use of the building blocks we described in previous sections to produce the final result.

A more thorough performance evaluation is given in Section 16.5.2.

## 14.2 OptiQL Struct Of Arrays Example

OptiQL is a DSL for data querying of in-memory collections, inspired by LINQ [90]. We consider querying a data set with roughly 10 columns, similar to the table lineItems from the TPCH benchmark. The example is slightly trimmed down from TPCH Query 1:

```
val res = lineItems Where(_.l_shipdate <= Date("1998-12-01"))
GroupBy(l => l.l_returnflag) Select(g => new Result {
  val returnFlag = g.key
  val sumQty = g.Sum(_.l_quantity)
})
```

A straightforward implementation is rather slow. There are multiple traversals that compute intermediate data structures. There is also a nested `Sum` operation inside the `Select` that follows the `groupBy`.

We can translate this code to a single while loop that does not construct any intermediate data structures and furthermore ignores all columns that are not part of the result. First, the complete computation is split into separate loops, one for each column. Unnecessary ones are removed. Then the remaining component loops are reassembled via loop fusion. For the full TPCH Query 1, these transformations provide a speed up of 5.5x single threaded and 8.7x with 8 threads over the baseline array-of-struct version (see Section 16.5).

We use two hash tables in slightly different ways: one to accumulate the keys (so it is really a hash set) and the other one to accumulate partial sums. Internally there is only one hash table that maps keys to positions. The partial sums are just kept in an array that shares the same indices with the key array.

Below is the annotated generated code:

```
  val x11 = x10.column("l_returnflag")
  val x20 = x10.column("l_shipdate")
  val x52 = generated.scala.util.Date("1998-12-01")
  val x16 = x10.columns("l_quantity")
  val x283 = x264 + x265

  // hash table constituents, grouped for both x304,x306
  var x304x306_hash_to_pos: Array[Int] = alloc_htable // actual hash table
  var x304x306_hash_keys: Array[Char] = alloc_buffer  // holds keys
  var x304_hash_data: Array[Char] = alloc_buffer      // first column data
  var x306_hash_data: Array[Double] = alloc_buffer    // second column data
  val x306_zero = 0.0
  var x33 = 0
  while (x33 < x28) {  // begin fat loop x304,x306
    val x35 = x11(x33)
    val x44 = x20(x33)
    val x53 = x44 <= x52
    val x40 = x16(x33)

    // group conditionals
```

```
    if (x53) {
      val x35_hash_val = x35.hashCode
      val x304x306_hash_index_x35 = {
        // code to lookup x35_hash_val
        // in hash table x304x306_hash_to_pos
        // with key table x304x306_hash_keys
        // (growing hash table if necessary)
      }

      if (x304x306_hash_index_x35 >= x304x306_hash_keys.length) { // not found
        // grow x304x306_hash_keys and add key
        // grow x304_hash_data
        // grow x306_hash_data and set to x306_zero
      }
      x304_hash_data (x304x306_hash_index_x35) = x35

      val x264 = x306_hash_data (x304x306_hash_index_x35)
      val x265 = x40
      val x283 = x264 + x265
      x304_hash_data (x304x306_hash_index_x35) = x283
    }
  } // end fat loop x304,x306
  val x304 = x304_hash_data
  val x305 = x304x306_hash_to_pos.size
  val x306 = x306_hash_data

  val x307 = Map("returnFlag"->x304,"sumQty"->x306) //Array Result
  val x308 = Map("data"->x307,"size"->x305) //DataTable
```

## 14.3 Fast Fourier Transform Example

We consider staging a fast fourier transform (FFT) algorithm. A staged FFT, implemented in MetaOCaml, has been presented by Kiselyov et al. [76] Their work is a very good example for how staging allows to transform a simple, unoptimized algorithm into an efficient program generator. Achieving this in the context of MetaOCaml, however, required restructuring the program into monadic style and adding a front-end layer for performing symbolic rewritings. Using our approach of just adding `Rep` types, we can go from the naive textbook-algorithm to the staged version (shown in Figure 14.1) by changing literally two lines of code:

```
trait FFT { this: Arith with Trig =>
  case class Complex(re: Rep[Double], im: Rep[Double])
  ...
}
```

All that is needed is adding the self-type annotation to import arithmetic and trigonometric operations and changing the type of the real and imaginary components of complex numbers from `Double` to `Rep[Double]`.

```
trait FFT { this: Arith with Trig =>
  case class Complex(re: Rep[Double], im: Rep[Double]) {
    def +(that: Complex) = Complex(this.re + that.re, this.im + that.im)
    def *(that: Complex) = ...
  }
  def omega(k: Int, N: Int): Complex = {
    val kth = -2.0 * k * Math.Pi / N
    Complex(cos(kth), sin(kth))
  }
  def fft(xs: Array[Complex]): Array[Complex] = xs match {
    case (x :: Nil) => xs
    case _ =>
      val N = xs.length // assume it's a power of two
      val (even0, odd0) = splitEvenOdd(xs)
      val (even1, odd1) = (fft(even0), fft(odd0))
      val (even2, odd2) = (even1 zip odd1 zipWithIndex) map {
        case ((x, y), k) =>
          val z = omega(k, N) * y
          (x + z, x - z)
      }.unzip;
      even2 ::: odd2
  }
}
```

Figure 14.1: FFT code. Only the real and imaginary components of complex numbers need to be staged.

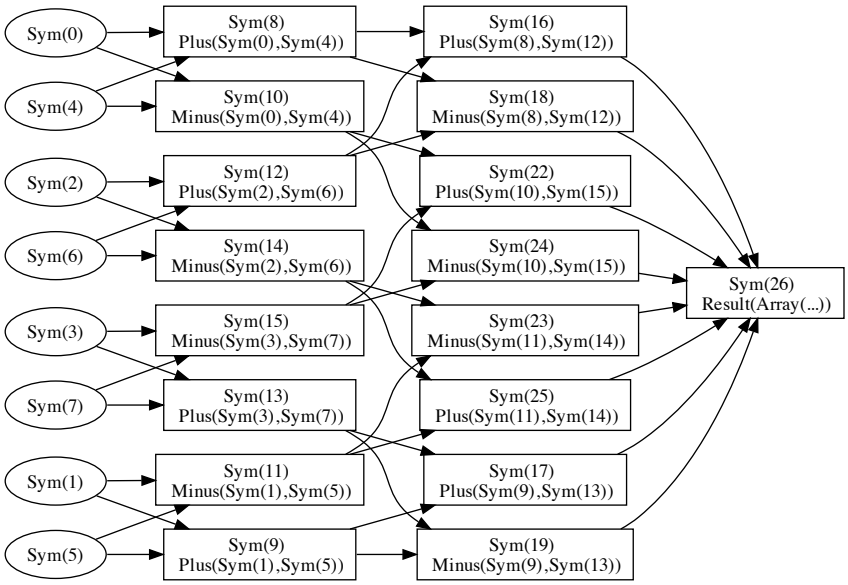



Figure 14.2: Computation graph for size-4 FFT. Auto-generated from staged code in Figure 14.1.

```
trait ArithExpOptFFT extends ArithExp {
  override def infix_*(x:Exp[Double],y:Exp[Double]) = (x,y) match {
    case (Const(k), Def(Times(Const(l), y))) => Const(k * l) * y
    case (x, Def(Times(Const(k), y))) => Const(k) * (x * y))
    case (Def(Times(Const(k), x)), y) => Const(k) * (x * y))
    ...
    case (x, Const(y)) => Times(Const(y), x)
    case _ => super.infix_*(x, y)
  }
}
```

Figure 14.3: Extending the generic implementation from Section 9.2.3 with FFT-specific optimizations.

Merely changing the types will not provide us with the desired optimizations yet. We will see below how we can add the transformations described by Kiselyov et al. to generate the same fixed-size FFT code, corresponding to the famous FFT butterfly networks (see Figure 14.2). Despite the seemingly naive algorithm, this staged code is free of branches, intermediate data structures and redundant computations. The important point here is that we can add these transformations without any further changes to the code in Figure 14.1, just by mixing in the trait `FFT` with a few others.

### 14.3.1 Implementing Optimizations

As already discussed in Section 9.2.3, some profitable optimizations are very generic (CSE, DCE, etc), whereas others are specific to the actual program. In the FFT case, Kiselyov et al. [76] describe a number of rewritings that are particularly effective for the patterns of code generated by the FFT algorithm but not as much for other programs.

What we want to achieve again is modularity, such that optimizations can be combined in a way that is most useful for a given task. This can be achieved by overriding smart constructors, as shown by trait `ArithExpOptFFT` (see Figure 14.3). Note that the use of `x*y` within the body of `infix_*` will apply the optimization recursively.

### 14.3.2 Running the Generated Code

Using the staged FFT implementation as part of some larger Scala program is straightforward but requires us to interface the generic algorithm with a concrete data representation. The algorithm in Figure 14.1 expects an array of `Complex` objects as input, each of which contains fields of type `Rep[Double]`. The algorithm itself has no notion of staged arrays but uses arrays only in the generator stage, which means that it is agnostic to how data is stored. The enclosing program, however, will store arrays of complex numbers in some native format which we will need to feed into the algorithm. A simple choice of representation is to use `Array[Double]` with the complex numbers flattened into adjacent slots. When applying `compile`, we will thus receive input of type `Rep[Array[Double]]`. Figure 14.4 shows how we can extend trait `FFT` to

```
trait FFTC extends FFT { this: Arrays with Compile =>
  def fftc(size: Int) = compile { input: Rep[Array[Double]] =>
    assert(<size is power of 2>) // happens at staging time
    val arg = Array.tabulate(size) { i =>
      Complex(input(2*i), input(2*i+1))
    }
    val res = fft(arg)
    updateArray(input, res.flatMap {
      case Complex(re,im) => Array(re,im)
    })
  }
}
```

Figure 14.4: Extending the FFT component from Figure 14.1 with explicit compilation.

FFTC to obtain compiled FFT implementations that realize the necessary data interface for a fixed input size.

We can then define code that creates and uses compiled FFT "codelets" by extending FFTC:

```
trait TestFFTC extends FFTC {
  val fft4: Array[Double] => Array[Double] = fftc(4)
  val fft8: Array[Double] => Array[Double] = fftc(8)

  // embedded code using fft4, fft8, ...
}
```

Constructing an instance of this subtrait (mixed in with the appropriate LMS traits) will execute the embedded code:

```
val OP: TestFFC = new TestFFTC with CompileScala
  with ArithExpOpt  with ArithExpOptFFT with ScalaGenArith
  with TrigExpOpt   with ScalaGenTrig
  with ArraysExpOpt with ScalaGenArrays
```

We can also use the compiled methods from outside the object:

```
OP.fft4(Array(1.0,0.0, 1.0,0.0, 2.0,0.0, 2.0,0.0))
↪ Array(6.0,0.0,-1.0,1.0,0.0,0.0,-1.0,-1.0)
```

Providing an explicit type in the definition `val OP: TestFFC = ...` ensures that the internal representation is not accessible from the outside, only the members defined by TestFFC.

## 14.4 Regular Expression Matcher Example

Specializing string matchers and parsers is a popular benchmark in the partial evaluation and supercompilation literature [26, 2, 124, 141, 123]. We consider "multi-threaded" regular expression matchers, that spawn a new conceptual thread to process alternatives in parallel. Of course these matchers do not actually spawn OS-level threads, but rather need to be advanced manually by client code. Thus, they are similar to coroutines.

Here is a simple example for the fixed regular expression `.*AAB`:

```
def findAAB(): NIO = {
  guard(Set('A')) {
    guard(Set('A')) {
      guard(Set('B'), Found)) {
        stop()
  }}} ++
  guard(None) { findAAB() } // in parallel...
}
```

We can easily add combinators on top of the core abstractions that take care of producing matchers from textual regular expressions. However the point here is to demonstrate how the implementation works.

The given matcher uses an API that models nondeterministic finite automata (NFA):

```
type NIO = List[Trans]   // state: many possible transitions
case class Trans(c: Set[Char], x: Flag, s: () => NIO)

def guard(cond: Set[Char], flag: Flag)(e: => NIO): NIO =
  List(Trans(cond, flag, () => e))
def stop(): NIO = Nil
```

An NFA state consists of a list of possible transitions. Each transition may be guarded by a set of characters and it may have a flag to be signaled if the transition is taken. It also knows how to compute the following state. We use `Char`s for simplicity, but of course we could use generic types as well. Note that the API does not mention where input is obtained from (files, streams, etc).

We will translate NFAs to DFAs using staging. This is the unstaged DFA API:

```
abstract class DfaState {
  def hasFlag(x: Flag): Boolean
  def next(c: Char): DfaState
}
def dfaFlagged(flag: Flag, link: DfaState) = new DfaState {
  def hasFlag(x: Flag) = x == flag || link.hasFlag(x)
  def next(c: Char) = link.next(c)
}
def dfaState(f: Char => DfaState) = new DfaState {
  def hasFlag(x: Flag) = false
  def next(c: Char) = f(c)
}
```

The staged API is just a thin wrapper:

```
type DIO = Rep[DfaState]
def dfa_flag(x: Flag)(link: DIO): DIO
def dfa_trans(f: Rep[Char] => DIO): DIO
```

Translating an NFA to a DFA is accomplished by creating a DFA state for each encountered NFA configuration (removing duplicate states via `canonicalize`):

```
def exploreNFA[A](xs: NIO, cin: Rep[Char])(flag: Flag => Rep[A] => Rep[A])
                                          (k: NIO => Rep[A]):Rep[A] = xs match {
  case Nil => k(Nil)
  case Trans(Set(c), e, s)::rest =>
    if (cin == c) {
      // found match: drop transitions that look for other chars and
      // remove redundant checks
      val xs1 = rest collect { case Trans(Set(‘c‘)|None,e,s) => Trans(Set(),e,s) }
      val maybeFlag = e map flag getOrElse (x=>x)
      maybeFlag(exploreNFA(xs1, cin)(acc => k(acc ++ s())))
    } else {
      // no match, drop transitions that look for same char
      val xs1 = rest filter { case Trans(Set(‘c‘),_,_) => false case _ => true }
      exploreNFA(xs1, cin)(k)
    }
  case Trans(Set(), e, s)::rest =>
    val maybeFlag = e map flag getOrElse (x=>x)
    maybeFlag(exploreNFA(rest, cin)(acc => k(acc ++ s())))
}
```

Figure 14.5: NFA Exploration

```
def convertNFAtoDFA(states: NIO): DIO = {
  val cstates = canonicalize(state)
  dfa_trans { c: Rep[Char] =>
    exploreNFA(cstates, c)(dfa_flag) { next =>
      convertNFAtoDFA(next)
    }
  }
}
iterate(findAAB())
```

The LMS framework memoizes functions (see Section 6.2.4) which ensures termination if the NFA is indeed finite.

We use a separate function to explore the NFA space (see Figure 14.5), advancing the automaton by a symbolic character `cin` to invoke its continuations `k` with a new automaton, i.e. the possible set of states after consuming `cin`. The given implementation assumes character sets contain either zero or one characters, the empty set `Set()` denoting a wildcard match. More elaborate cases such as character ranges are easy to add. The algorithm tries to remove as many redundant checks and impossible branches as possible. This only works because the character guards are staging time values.

The generated code is shown in Figure 14.6. Each function corresponds to one DFA state. Note how negative information has been used to prune the transition space: Given input such as `...AAB` the automaton jumps back to the initial state, i.e. it recognizes that the last character B cannot also be A and starts looking for two As after the B.

```
def stagedFindAAB(): DfaState = {
  val x7 = { x8: (Char) =>
    // matched AA
    val x9 = x8 == B
    val x15 = if (x9) {
      x11
    } else {
      val x12 = x8 == A
      val x14 = if (x12) {
        x13
      } else {
        x10
      }
      x14
    }
    x15
  }
  val x13 = dfaState(x7)
  val x4 = { x5: (Char) =>
    // matched A
    val x6 = x5 == A
    val x16 = if (x6) {
      x13
    } else {
      x10
    }
    x16
  }
  val x17 = dfaState(x4)
  val x1 = { x2: (Char) =>
    // matched nothing
    val x3 = x2 == A
    val x18 = if (x3) {
      x17
    } else {
      x10
    }
    x18
  }
  val x10 = dfaState(x1)
  val x11 = dfaFlagged(Found, x10)
  x10
}
```

Figure 14.6: Generated matcher code for regular expression `.*AAB`

The generated code can be used as follows:

```
var state = stagedFindAAB()
var input = ...
while (input.nonEmpty) {
  state = state.next(input.head)
  if (state.hasFlag(Found))
    println("found AAB. rest: " + input.tail)
  input = input.tail
}
```

If the matcher and input iteration logic is generated together, further translations can be applied to transform the mutually recursive lambdas into tight imperative state machines.[2]

[2] *Credits:* Optimizations implemented by Nada Amin

# Part IV

# Validation and Evaluation

# Chapter 15

# Intro

This part present projects undertaken that validate the use of LMS and embedded compilers.

**Disclosure** These projects are large efforts driven by many people. The author of this thesis has co-authored publications on each of the projects. Most of the discussed aspects are not to be understood as contributions of this thesis but as examples of how the research presented earlier in this thesis has been used in large projects and other people's works.

# Chapter 16

# Delite

[1] Delite [11, 113, 83] is a research project lead by Stanford University's Pervasive Parallelism Laboratory (PPL). Delite is a compiler framework and runtime for parallel embedded domain-specific languages (DSLs). To enable the rapid construction of high-performance, highly productive DSLs, Delite provides several facilities:

- Code generators for Scala, C++ and CUDA
- Built-in parallel execution patterns
- Optimizers for parallel code
- A DSL runtime for heterogeneous hardware

The material in this chapter is taken from [83, 127][2]. The Delite Compiler Framework is capable of expressing parallelism both within and among DSL operations, as well as performing useful parallel analyses. It also provides a framework for adding domain-specific optimizations. It then generates a machine-agnostic intermediate representation of the program which is consumed by the Delite Runtime. The runtime system provides a common set of features required by most DSLs, such as scheduling work across hardware resources and managing communication.

## 16.1 Building Parallel DSLs Using Delite

### 16.1.1 Building an Intermediate Representation (IR)

In Delite, a single IR node can be viewed from three different perspectives (as depicted in Figure 16.1) which provide different optimizations and code generation strategies. Delite is built using the concept of a multi-view IR.

**Generic IR:** The most basic view of an IR node is a symbol and its definition, which is similar to a node in the flow graph of a traditional compiler framework. Therefore, we

[1] *Credits:* Joint work with Arvind Sujeeth, Kevin Brown, HyoukJoong Lee, Hassan Chafi, Kunle Olukotun
[2] *Credits:* Presentation HyoukJoong Lee, Arvind Sujeeth, Kevin Brown

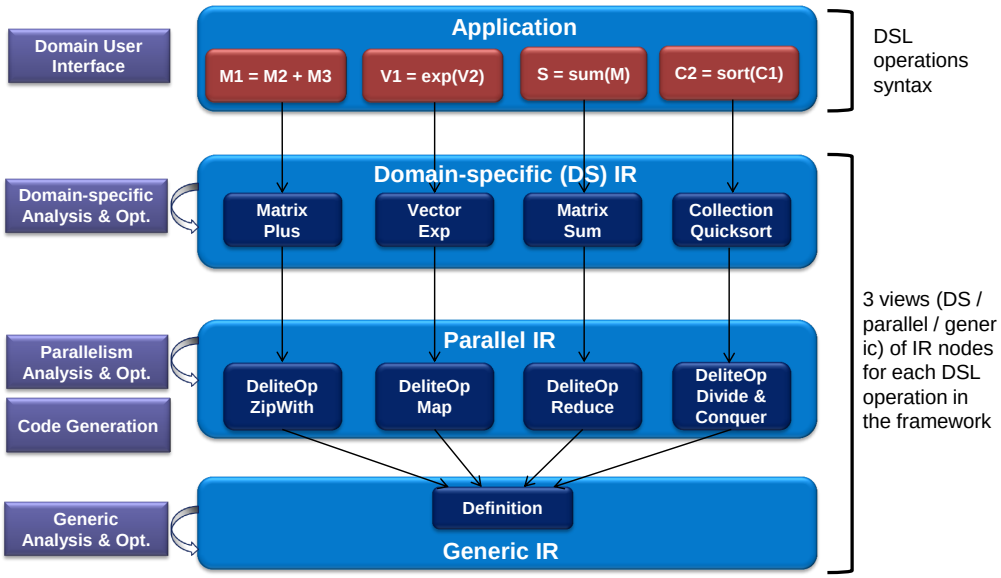



Figure 16.1: Multi-view of IR nodes in the Delite Compilation Framework. For example, the matrix addition operation (`M1 = M2 + M3`) is represented as a `MatrixPlus` IR node, which extends the `DeliteOpZipWith` IR node which again extends the `Definition` IR node. The generic IR view is used for traditional compiler optimizations, the parallel IR view is used for exposing parallel patterns and loop fusing optimizations, and the Domain-specific IR view is used for domain-specific optimizations.

can apply all the well-known static optimizations at this level. The primary difference is that our representation has a coarser granularity because each node is a DSL operation rather than an individual instruction, and this often leads to better optimization results. For example, the common subexpression elimination (CSE) can be applied to the vector operations $(x(i) - mu0, x(i) - mu1)$ as shown in Figure 16.2 instead of just to scalar operations. Currently applied optimizations include CSE, constant propagation, dead code elimination, and code motion.

**Parallel IR:** A generic IR node can be characterized by its parallel execution pattern. At this level of view, the Delite Compiler Framework provides a finite number of common structured parallel execution patterns in the form of `DeliteOp` IR nodes. Examples include `DeliteOpMap` which encodes disjoint element access patterns without ordering constraints, and `DeliteOpForeach` which allows a DSL-defined consistency model for overlapping elements. The `DeliteOpSequential` IR node is for the pattern that is not parallelizable. Since certain parallel patterns share a common notion of loops, multiple loop patterns can be fused into a single loop. The parallel IR optimizer iterates over all of the IR nodes of loop types (e.g., `DeliteOpMap`, `DeliteOpZipwith`, etc.), and fuses those with the same number of iterations into a single loop. This optimization removes unnecessary memory allocations and also improves cache behavior by eliminating multiple passes over data, which is especially useful for memory-bound applications.

**Domain-specific (DS) IR:** Since the parallel IR does not encode domain-specific infor-

mation, there is another viewpoint for semantic information of the operation. This enables domain-specific optimizations such as linear algebra simplification. The transformation rules are simply described by pattern matching on DS IR nodes, and the optimizer replaces the matched nodes with a simpler set of nodes. Examples on matrix operations are $(A^t)^t = A$, and $A*B+A*C = A*(B+C)$. Since the DSL developer has expertise in the execution patterns of each DS IR node, the DSL developer extends the appropriate Delite parallel IR node. In this way parallel execution patterns are abstracted away from DSL users.

This multi-view IR greatly simplifies the process of developing a new DSL since the generic IR and the parallel IR can be re-used by all DSLs, and therefore DSL developers only need to design a DS IR for each operation as an extension. In other words, DSL developers are only exposed to a high-level parallel instruction set (the parallel IR nodes) and the implementation details of each pattern on multiple targets are automatically managed by the Delite Compiler Framework.

Delite uses LMS to build the IR from DSL applications As the application starts executing within the framework, each operation is lifted to a symbolic representation to form an IR node rather than actually being executed. The IR nodes track all dependencies among one another and the various optimizations mentioned above are applied. After building the machine-independent IR, the Delite Compiler Framework starts the code generation phase to target heterogeneous parallel hardware.

### 16.1.2 Heterogeneous Target Code Generation

Generating a single binary executable for the application at compile time limits the portability of the application and requires runtime and hardware systems to rediscover dependency information in order to make machine-specific scheduling decisions. The Delite Compiler Framework defers such decisions by generating kernels for each IR node in multiple target programming models as well as the Delite Execution Graph describing the dependencies among kernels. Currently supported targets are Scala [99], C++, and CUDA.

#### Delite Execution Graph (DEG)

The Delite generator is the main code generator that controls multiple target generators. It first schedules IR nodes to form kernels in the execution graph, and iterates over the list of available target generators to generate corresponding target code for the kernel. It may not be possible to generate the kernel for all targets, but the kernel generation will succeed as long as at least one target succeeds. The fact that each IR node has multiple viewpoints means that they can also be generated in different ways for each view. For example, a matrix addition kernel could be generated in the domain-specific view written by the DSL developer, but it also can be generated in the parallel view since the operation is of type `DeliteOpZipWith`. Since the Delite Compiler Framework provides parallel implementations for `DeliteOps`, DSL developers do not have to provide code generators when they extend one of the parallel IR nodes. When DSL developers already have an efficient implementation of the kernel (e.g., BLAS libraries for matrix multiplication), they can generate calls to the external library using `DeliteOpExternal`.

#### GPU code generation

GPU code generation requires additional work since the programming model has more constraints compared to the Scala and C++ targets. One major issue is memory allocation. Since dynamic memory allocation within the kernel is either not possible or not practical for performance in GPU programming models, all device memory allocations within the kernel are pre-allocated by the Delite Runtime before launching the kernel. This is enabled by the CUDA generator collecting the memory requirement information and passing it to the runtime through a metadata field in the DEG. In addition, since the GPU resides in a separate address space, input/output transfer functions are also generated so that the Delite Runtime can manage data communication. Kernel configuration information (the dimensionality and the size of each dimension) also needs to be generated by the CUDA generator.

#### Variants

When multiple data parallel operations are nested, various parallelization strategies exist. In a simple case, a `DeliteOpMap` op within a `DeliteOpMap` can parallelize the outer loop or the inner loop or both. Therefore, the Delite Compiler Framework generates a data parallel operation in both a sequential version and a parallel version to provide flexible parallelization options when they are nested. This feature is especially useful for the CUDA target generator to improve the coverage of GPU kernels, since parallelizing the outer loop is not always possible for GPU due to the memory allocation requirements of the kernel. In those cases, the outer loop is serialized and only the inner loop is parallelized as a GPU kernel.

#### Target-specific Optimizations

While machine-independent optimizations are applied when building the IR, machine-specific optimizations are applied during the code generation phase. For example, the memory access patterns that enable better bandwidth utilization may not always be the same on the CPU and the GPU. Consider a data-parallel operation on each row of a matrix stored in a row-major format. In the case of the CPU where each core has its own private cache, assigning each row to each core naturally exploits spatial cache locality and prevents false sharing. However, the GPU prefers the opposite access pattern where each thread accesses each column, because the memory controller can coalesce requests from adjacent threads into a single transfer. Therefore the CUDA generator emits code that uses a transposed matrix with inverted indices for efficient GPU execution. In addition, to exploit SIMD units for data-parallel operations on the CPU, we currently generate source code that can be vectorized by the target compiler. It would also be straightforward to generate explicit SIMD instructions (e.g., SSE, AVX).

## 16.2 Executing Embedded Parallel DSLs

DSLs targeting heterogeneous parallelism require a runtime to manage application execution. The work done at this phase of execution includes generating a large amount of "plumbing"

code focused on managing parallel execution on a specific parallel architecture. The implementation can be difficult to get right, both in terms of correctness and efficiency, but is common across DSLs. We therefore built a heterogeneous parallel runtime to provide these shared services for all Delite DSLs.

### 16.2.1 Scheduling the Delite Execution Graph (DEG)

The Delite Runtime combines the machine-agnostic DEG generated by the framework with the specification of the current machine (e.g., number of CPUs, number of GPUs, etc.) to schedule the application across the available hardware resources. The Delite Runtime schedules the application before beginning execution using the static knowledge provided in the DEG. Since branch directions are still unknown, the Delite Runtime generates a partial schedule for every straight-line path in the application and resolves how to execute those schedules during execution. The scheduling algorithm attempts to minimize communication among kernels by scheduling dependent kernels on the same hardware resource and makes device decisions based on kernel and hardware availability. Sequential kernels are scheduled on a single resource while data-parallel kernels selected for CPU execution are split by the scheduler to execute on multiple hardware resources simultaneously. Since the best strategy for parallelizing and synchronizing these data-parallel chunks is not known until after scheduling, the runtime is responsible for generating the decomposition. In the case of a Reduce kernel, for example, the framework's code generator emits the reduction function and the runtime generates a tree-reduction implementation that is specialized to the number of processors selected to perform the reduction.

### 16.2.2 Generating Execution Plans for Each Hardware Resource

Dynamically dispatching kernels into a thread pool can have very high overheads. However, the knowledge provided by the DEG and static schedule of the application is sufficient to generate and compile an executable (execution plan) for each hardware resource. Each executable launches the kernels and performs the necessary synchronization for its resource according to the partial schedules. The combination of generating custom executables for the chosen schedule and delaying the injection of synchronization code until after scheduling allows for multiple optimizations in the compiled schedule that minimize runtime overhead. For example, data that does not escape a given resource does not require any synchronization. This synchronization code is customized to the underlying memory model between the communicating resources. When shared memory is available, the implementation simply passes the necessary pointers, and when the resources reside in separate address spaces it performs the necessary data transfers. Minimizing runtime overhead by eliminating unnecessary synchronization and removing the central kernel dispatch bottleneck enables applications to scale with much less work per kernel.

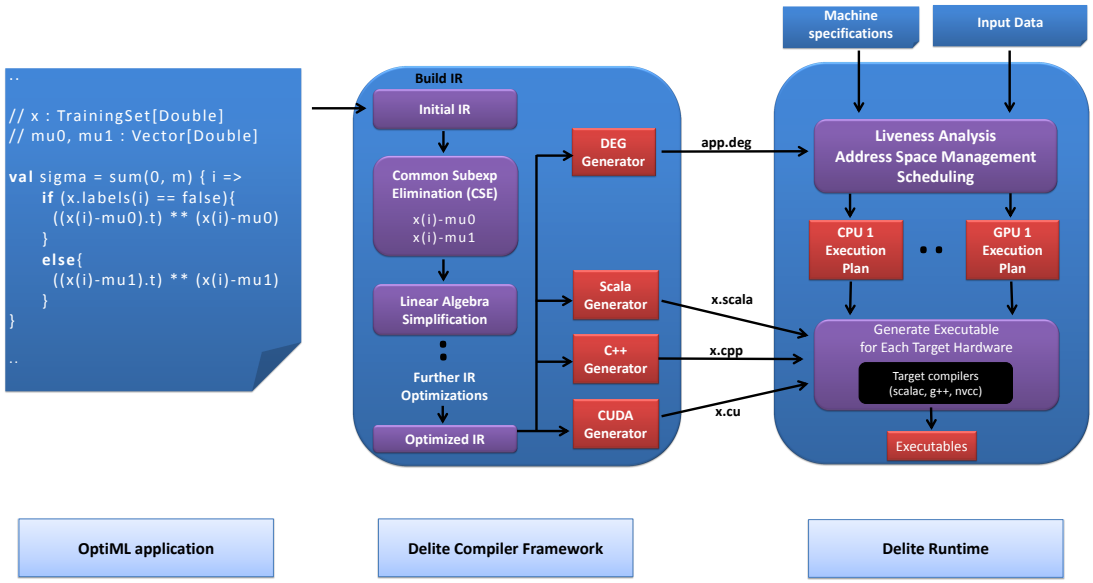



Figure 16.2: Entire process of the Delite Compiler Framework and Runtime

### 16.2.3 Managing Execution on Heterogeneous Parallel Hardware

Executing on heterogeneous hardware introduces new and difficult challenges compared to traditional uniprocessor or even multi-core systems. The introduction of multiple address spaces requires expensive data transfers that should be minimized. The Delite Runtime achieves this through detailed kernel dependency information provided by the DEG. The graph specifies which inputs a kernel will only read and which it will mutate. This information combined with the schedule allows the runtime to determine at any given time during the execution if the version of an input data structure in a given address space is currently nonexistent, valid, or old.

Managing the memory in each of these address spaces is also critical. The Delite Runtime currently utilizes the JVM to perform memory management for all CPU kernels, but GPUs have no such facilities. In addition, all memory used by a GPU kernel must be allocated prior to launching the kernel. In order to deal with these issues, the Delite Runtime pre-allocates all the data structures for a given GPU kernel by using the allocation information supplied by the framework's GPU code generator. The runtime also performs liveness analysis using the schedule to determine the earliest point at which each kernel's inputs and outputs are no longer needed by the GPU. By default the GPU host thread attempts to run ahead as much as possible, but when this creates memory pressure it uses the liveness information to wait until enough data becomes dead, free it, and continue executing.

## 16.3 Putting It All Together

The overall operation of the Delite Compiler Framework and Runtime is shown in Figure 16.2. The *sum* construct in the OptiML snippet accumulates the result of the given block for every iteration (*0* to *m*) and is implemented by extending the `DeliteOpMapReduce` parallel pattern of the framework.

To generate optimized executables from the high level representation of the DSL operations, the Delite compiler builds an intermediate representation (IR) of the application and applies various optimizations on the IR nodes. For example, since the two vector minus operations ($x(i) - mu0$) within the *sum* are redundant, common subexpression elimination removes the latter operation by reusing the former result. After building the IR, the code generators in the framework automatically emit computation kernels for both the CPU and the GPU. When all of the kernels of the application have been generated along with the Delite Execution Graph (DEG), which encodes the data and control dependencies of the kernels, the Delite Runtime starts analyzing the DEG to make scheduling decisions to generate execution plans. Necessary memory transfers and synchronization are added as the execution plan for each target is generated. Finally, the kernels from the compiler and the execution plan from the runtime are compiled and linked together by target language compilers (e.g., Scala, Cuda) to generate an executable that runs on the system.

## 16.4 Delite DSLs

A number of DSLs have been developed using Delite, some of which are presented below.

### 16.4.1 OptiML

OptiML is a DSL for machine learning designed for statistical inference problems expressed with vectors, matrices, and graphs [126]. OptiML supports linear algebra operations on mutable objects but restricts when side effects may be used, preventing many common parallel programming pitfalls by construction. For example, most OptiML language constructs (e.g. *sum*) do not allow side effects to be used within their anonymous function arguments. OptiML performs domain-specific optimizations such as linear algebra rewriting to simplify coarse-grained computation on vectors and matrices. Listing 16.1 shows an example snippet of OptiML code from the Naive Bayes algorithm. The `(n::m){}` statement constructs a vector by mapping an index *i* to a corresponding result value. This snippet computes the frequency of spam words in a corpus of emails. The key challenges with compiling OptiML for performance are reasoning about mutable aliases (to enable generic static optimizations) and an object-oriented inheritance hierarchy, which must still be efficiently targeted to multiple back-ends.

### 16.4.2 OptiQL

OptiQL is a DSL for data querying of in-memory collections. OptiQL is heavily inspired by LINQ [90], specifically LINQ to Objects. Listing 16.2 shows an example snippet of OptiQL code

```
// ts: TrainingSet[Double]
// returns a vector (phi_y1) of numTokens length,
// corresponding to each return value
val phi_y1 = (0::numTokens) { j =>
  val spamwordcount =
    sumIf(0, numTrainDocs) { ts.labels(_) == 1 }
                           { i => ts.t(j,i) }
  val spam_totalwords =
    sumIf(0, numTrainDocs) { ts.labels(_) == 1 }
                           { i => words_per_email(i) }
  (spamwordcount + 1) / (spam_totalwords + numTokens)
}
```

Listing 16.1: Naive Bayes snippet in OptiML

```
// lineItems: Iterable[LineItem]
// Similar to Q1 of the TPC-H benchmark
val q = lineItems Where(_.l_shipdate <= Date(''1998-12-01'')).
GroupBy(l => (l.l_linestatus)).
Select(g => new Result {
  val lineStatus = g.key
  val sumQty = g.Sum(_.l_quantity)
  val sumDiscountedPrice =
    g.Sum(l => l.l_extendedprice*(1.0-l.l_discount))
  val avgPrice = g.Average(_.l_extendedprice)
  val countOrder = g.Count
}) OrderBy(_.returnFlag) ThenBy(_.lineStatus)
```

Listing 16.2: Sample OptiQL query

that expresses a query similar to Q1 in the TPC-H benchmark. The query first excludes any line item objects with a ship date that occurs after the specified date. It then groups each line item by both its line status. Finally, it summarizes each group by aggregating the group's line items and constructs a final result object per group. The main challenge for OptiQL is to support user defined classes without giving up the ability to reason about how each field of the final structure has been constructed. Section 13 shows how extensions to the host language compiler help in overcoming this challenge.

### 16.4.3 OptiMesh

OptiMesh is an implementation of Liszt [35] on Delite. Liszt is a DSL for mesh-based partial differential equation solvers. Liszt code allows users to perform iterative computation over mesh elements (e.g. cells, faces). Data associated with mesh elements are stored in external fields that are indexed by the elements. Listing 16.3 shows a simple OptiMesh program that computes the flux through edges in the mesh. As the snippet demonstrates, a key challenge with OptiMesh is to detect write conflicts within for comprehensions given a particular mesh input. We load the program input at staging time and build an interference graph for each

```
for (edge <- edges(mesh)) {
  val flux = flux_calc(edge)
  val v0 = head(edge)
  val v1 = tail(edge)
  Flux(v0) += flux // possible write conflicts!
  Flux(v1) -= flux
}
```

Listing 16.3: Sample OptiMesh program

```
for(t <- G.Nodes) {
  val Val = ((1.0 - d) / N) +
    d * Sum(t.InNbrs){w => PR(w) / w.OutDegree}
  PR <= (t,Val)
  diff += Math.abs(Val - PR(t))
}
```

Listing 16.4: Core of the PageRank algorithm in OptiGraph

mesh comprehension through a symbolic execution of the program. The interference graph is then used to color a single for comprehension into a set of non-conflicting loops. While Liszt is implemented in a standalone DSL, OptiMesh performs the same analyses and transformations within the Delite environment.

### 16.4.4 OptiGraph

OptiGraph is a DSL for static graph analysis based on the Green-Marl DSL [57]. OptiGraph defines types for directed and undirected graphs, nodes, and edges. It allows data to be associated with graph nodes and edges via node and edge property types and provides three types of collections for node and edge storage (namely, `Set`, `Sequence`, and `Order`). Furthermore, OptiGraph defines constructs for `BFS` and `DFS` order graph traversal, sequential and explicitly parallel iteration, and implicitly parallel in-place reductions and group assignments. An important feature of OptiGraph is also its built-in support for bulk synchronous consistency via *deferred assignments*.

Listing 16.4 shows the parallel loop of the PageRank algorithm [102] written in OptiGraph. `PR` is a node property associating a page-rank value with every node in the graph. The `<=` statement is a deferred assignment of the new page-rank value, `Val`, for node `t`; deferred writes to `PR` are made visible after the **`for`** loop completes via an explicit assignment statement (not shown). Similarly, `+=` is a scalar reduction that implicitly writes to `diff` only after the loop completes. In contrast, `Sum` is an in-place reduction over the parents of node `t`.

### 16.4.5 OptiCollections

While not strictly a DSL, OptiCollections is an example of the optimizing compilation techniques applied to the Scala collections library. A key benefit of providing OptiCollections as

```
OptiCollections {
  val sourcedests = pagelinks flatMap { l =>
    val sd = l.split(":")
    val source = Long.parseLong(sd(0))
    val dests = sd(1).trim.split(" ")
    dests.map(d => (Integer.parseInt(d), source))
  }
  val inverted = sourcedests groupBy (x => x._1)
}
```

Listing 16.5: OptiCollections example

a module is that it can be mixed to enrich other DSLs with a range of collection types and operations on those types. Finally, the OptiCollections module can also be used within regular Scala programs to speed up their performance critical parts. The reverse-web link example presented in Listing 16.5 shows that OptiCollections can be used transparently – there is no difference in syntax compared to the same snippet written using Scala collections.

### 16.4.6 DSL Extensibility

Delite DSLs can be explicitly extended by a DSL author and new DSLs can be composed out of existing ones. Although OptiML was originally conceived of as a monolithic DSL for machine learning, we have found that several DSLs need some linear algebra capability. Therefore, we refactored OptiML into an extension of a common linear algebra DSL, OptiLA. OptiML integrates tightly with OptiLA by inheriting all of its data types and operations and then extending existing types (e.g. `Vector`) with new operations or adding new abstractions (e.g. `TrainingSet`).The OptiML-OptiLA inheritance is simple to achieve with Delite's modular architecture and Scala trait mix-in inheritance. Consider the following definition for OptiLA operations:

```
trait OptiLA extends OptiLAScalaOpsPkg with LanguageOps
with ArithOps with VectorOps // with ...
```

OptiML extends OptiLA simply by extending the relevant packages:

```
trait OptiML extends OptiLA with OptiMLScalaOpsPkg
with OptiMLVectorOps with StreamOps // with ...
```

OptiML is a superset of OptiLA, but OptiML users only interact with OptiML and do not need to be aware of the existence of OptiLA. By extracting the linear algebra portion of OptiML into a separate DSL, we can reuse it for other DSLs; we are currently investigating using OptiLA as the basis of a convex optimization DSL.

Similarly, the OptiCollections module can be mixed in by other Delite DSLs by including the relevant traits in the DSL definition. When DSL users invoke an OptiCollections operation, that operation will construct an OptiCollections IR node. Note that after mix-in, the result is effectively a single DSL that extends common IR nodes; optimizations that operate on the generic IR can occur even between operations from different DSLs. This is analagous to

```
val orderData = DeliteRef[Array[Record]]()
val theta = DeliteRef[Array[Double]]()
OptiQL {
  // customers: Array[Customer], orders: Array[Order]
  val orders = customers Join(orders)
    WhereEq(_.c_custkey, _.o_custkey)
    Select((c,o) => new Result {
      val nationKey = c.c_nationkey
      val acctBalance = c.c_acctbal
      val price = o.o_totalprice
    })
  orderData.set(orders)
}
OptiML {
  // run linear regression on price
  val data = Matrix(orderData.get.map(t =>
    Array(t.getDouble(1),t.getDouble(2),t.getDouble(3))))
  val x = data.sliceCols(0,1)
  val y = data.getCol(2)
  theta.set(linreg.weighted(x,y).toArray)
}
println("theta: " + theta.get)
```

Listing 16.6: Interoperability example with OptiML and OptiQL

libraries building upon other libraries, except that now optimizing compilation can also be inherited. DSLs can add analyses and transformations that are designed to be included by other DSLs. The trade-off is that DSL authors must be aware of the semantics they are composing and are responsible for ensuring that transformations and optimizations between DSLs retain their original semantics. Namespacing can also be a pitfall and requires conventions like DSL specific prefixes (e.g. `optiml_vector_plus`). DSL traits cannot have conflicting object or method names since they are mixed in to the same object.

### 16.4.7 DSL Interoperability

DSL users developing large applications will often need to use multiple DSLs or fall back to a general-purpose language for non performance-critical tasks. Unlike interoperating with pure libraries, DSLs need a degree of isolation to perform analyses and enforce restricted semantics. We introduce coarse-grained execution blocks called *scopes* to encapsulate DSL code sections inside the host language (Scala). Consider the example program in Listing 16.6 which uses both OptiML and OptiQL. The example constructs a join of Order and Customer data, selecting a few of their attributes, and then runs linear regression on the total price of the orders.

The OptiML block construct is defined as follows:

```
def OptiML(b: => Unit) =
  new Scope(OptiMLApp, OptiMLAppRunner, Unit)(b)
```

OptiMLApp and OptiMLAppRunner are Scala traits that define the DSL interface and its implementation, respectively. The scala-virtualized compiler transforms function calls with return type `Scope` into an object that composes the two traits with the given block, making all members of the DSL interface available to the block's content. The implementation of the DSL interface remains hidden, however, to ensure safe encapsulation. The object's constructor then executes the block. The result is that each `Scope` is staged and executed independently, but can communicate inputs and outputs through heap objects (e.g. `DeliteRef` in Listing 16.6). This mechanism also allows DSL blocks to be embedded inside a normal Scala program (with the caveat that each `Scope` may take a non-trivial amount of time to stage and execute).

## 16.5 Performance Evaluation

We compare application performance for each DSL compared to the performance of the same application in existing alternative programming environments and analyze the benefits of the Delite compiler's various optimizations. For OptiML, we compare against explicitly parallelized MATLAB 7.11 [88] (using the Parallel Computing Toolbox), MATLAB's GPU computing support, as well as hand-optimized C for a sequential baseline.

We compare OptiQL and OptiCollections against Scala Parallel Collections, a module in the Scala standard library which provides data-parallel bulk operations [107].

We compare OptiGraph and OptiMesh to Green-Marl and Liszt respectively, stand-alone DSLs designed for high performance on heterogeneous hardware that have been shown to be competitive with optimized C++ implementations [57, 35]. In addition, Liszt is also capable of targeting both multicore CPU and GPU transparently, using a completely custom Liszt specific compiler that generates C++ and Cuda.

All of our experiments were performed on a Dell Precision T7500n with two quad-core Xeon 2.67GHz processors, 96GB of RAM, and an NVidia Tesla C2050. For the CPU each Delite DSL generated Scala code executed on the Oracle Java SE Runtime Environment 1.7.0 and the Hotspot 64-bit server VM with default options. For the GPU, Delite generated and executed Cuda v4.0. We ran each application ten times (to warm up the JIT) and report the average of the last 5 runs. For each run we timed the computational portion of the application. For each application we show normalized execution time relative to our DSL version with the speedup listed at the top of each bar.

**OptiML: k-means** The first OptiML application is k-means clustering on a 1,048,576 pixel by 3 color component image. We present OptiML performance with and without optimizations (Figure 16.3). By turning off both loop fusion and pattern-based rewrite rules the OptiML implementation becomes about 3.9x slower. In addition, enabling only one of the optimizations produces very modest performance improvements. This is due to the fact that the OptiML version of k-means relies heavily on loop fusion to obtain high performance and Delite's loop fusion optimizer relies on specific rewrites to uncover more loops that are eligible for fusing. The OptiML versions scale well from one to eight cores. In contrast, the MATLAB version does not scale well because we were forced to choose between vectorization and parallelization of

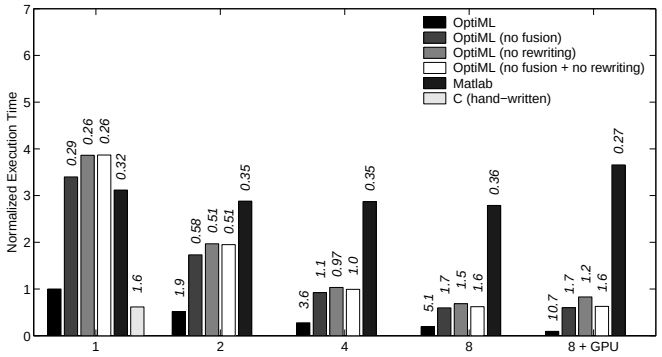



Figure 16.3: Performance of OptiML: kmeans.

the loops and the vectorized version was faster even at 8 cores than the more parallel version. Finally we implemented a hand-optimized sequential C version of k-means which includes optimizations such as aggressive memory reuse which make the application unparallelizable. This version is faster than OptiML single-threaded, but the OptiML version is parallel and surpasses the C version with only 2 cores. For the GPU version we wrote a separate MATLAB version using its GPU support. The OptiML version is able to transparently expand to 8 cores + 1 GPU to achieve speedup beyond the multicore versions. The unoptimized OptiML versions do not speed up with the GPU as the loop fusion optimization enables Delite to generate CUDA implementations for critical kernels in the application.

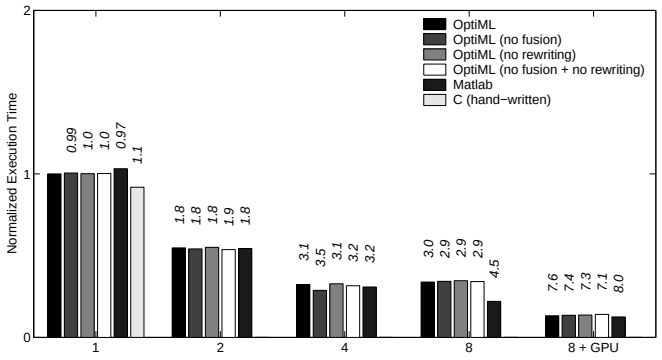



Figure 16.4: Performance of OptiML: rbm.

**OptiML: RBM** The second OptiML benchmark application is a Restricted Boltzmann machine with 2,000 hidden units and 2,000 dimensions. This application is dominated by matrix multiplication and other matrix-vector operations which can be executed efficiently using BLAS. Due to these application features OptiML's optimizations have virtually no impact on the execution time (Figure 16.4). In addition, the MATLAB muticore and GPU versions compare very well to the OptiML versions, and the C version is only slightly faster single-threaded as all versions of the application spend a large percentage of their execution time in BLAS library calls. Running with 8 cores + 1 GPU provides almost no improvement over 1 core + 1

GPU as all of the time-consuming operations can be shipped to the GPU, leaving very little work for the CPU to parallelize.

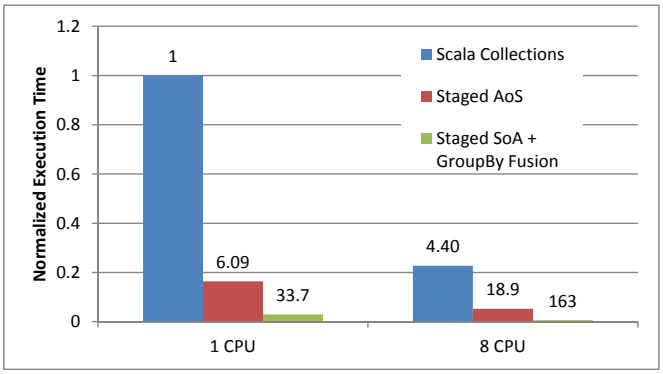



Figure 16.5: Performance of OptiQL: TPC-H Q1, AoS and SoA.

**OptiQL: TPC-H query 1** Figure 16.5 compares the performance of TPC-H query 1 on OptiQL vs. Scala Collections. When enabling array of structs to structs of array transformations, we gained an additional speedup of rougly 5.5x over the baseline OptiQL version for 1 thread and roughly 8.7x speedup over 8-thread baseline OptiQL. The total speedup of OptiQL 8-thread over Scala Collections 1-thread is 163x.

We experimented with shipping filters to the GPU and found that for these queries the time required to move the input data to the GPU was greater than the time for a single CPU to compute the result. To overcome the transfer overhead, the queries must contain enough computation to hide the transfer latency. However, typical OptiQL queries have too little arithmetic intensity to satisfy this requirement.

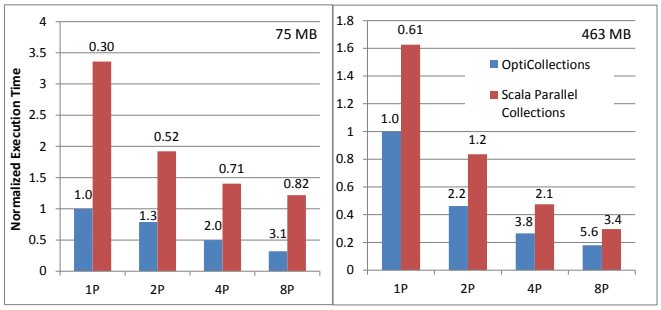



Figure 16.6: Performance of OptiCollections: reverse web-link graph.

**OptiCollections: reverse web-link graph** The reverse web-link graph program consumes a list of webpages and their outgoing links and outputs a list of webpages and the set of incoming links for each of the pages. The OptiCollections version is significantly faster at all thread counts and scales better with larger datasets, illustrating the benefit of staged compilation compared to the straight library approach employed by Scala parallel collections.

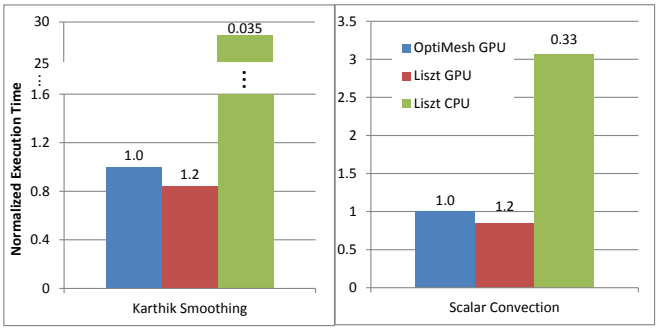



Figure 16.7: Performance of OptiMesh: scalar convection and Karthik smoothing.

**OptiMesh: scalar convection and Karthik smoothing** For both OptiMesh applications, Delite is able to generate and execute a Cuda kernel for each colored foreach loop, and achieve performance comparable to the Liszt GPU implementations for both applications (Figure 16.7). The GPU implementation is significantly faster than the Liszt sequential CPU version, which is written in C++. These applications perform very well on the GPU due to the large amount of compute and parallel nature of the operations. These applications utilize trigonometric functions heavily (in particular arctangent and arccosine), and therefore JVM implementations are not competitive with native implementations due to the fact that C++ library implementations use custom hardware instructions, while the Java library implementations are software-based in order to maintain platform independence.

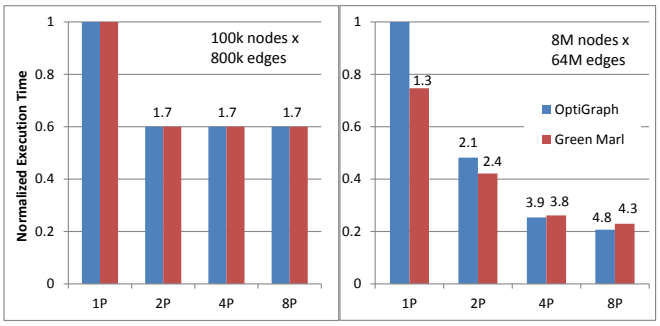


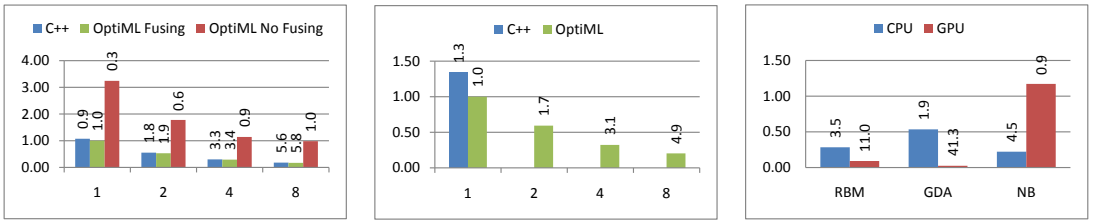

Figure 16.8: Performance of OptiGraph: page rank.

**OptiGraph: page rank** Figure 16.8 compares the performance of the PageRank algorithm [102] implemented in OptiGraph to the Green-Marl implementation on two different uniform random graphs of sizes 100k nodes by 800k edges and 8M nodes by 64M edges, respectively. This benchmark is dominated by the random memory accesses during node neighborhood exploration. Since OptiGraph's memory access patterns and the memory layout of its back-end data-structures are similar to those of Green-Marl, OptiGraph's sequential performance and scalability across multiple processors is close to that of Green-Marl. The back-end support for GPUs is currently being developed for both OptiGraph and Green-Marl to take advantage of recent advances in GPU graph processing [56], [91].

### 16.5.1 Discussion

From the above set of DSLs and applications we see a number of important trends. First of all, the Delite compiler's ability to orchestrate multiple powerful optimizations such as loop fusion and pattern-based rewrite rules allows for peformance comparable to optimized low-level versions and significantly faster than alternative high-level library implementations, as seen for OpiML, OptiQL, and OptiCollections. As seen in the OptiML k-means study of disabling loop fusion, simple methods of removing abstraction and launching optimized kernels naively from the application are not always sufficient for obtaining performance compared to low-level hand-optimized implementations. Rather, high-level optimizations are necessary to convert the application-level description into an efficient implementation for the underlying hardware. In addition, it is critical to be able to target heterogeneous environments for maximum performance. For OptiQL, the nature of the datasets and operations made GPU execution prohibitively expensive, while for OptiMesh the GPU produces results over 30x faster than a single CPU core. The best choice of execution hardware cannot always be chosen universally for a given DSL either. For OptiML, RBM runs faster using only the GPU, but k-means is faster with 8 cores than GPU only. However, by scheduling some kernels across multiple CPU threads and some kernels on the GPU, the application sped up overall. Since it is often unclear in advance just how well an application will run on a given hardware target it is important for programmer productivity to be able to target multicore execution, GPU execution, and multicore + GPU execution from a single appication source.

### 16.5.2 OptiML vs C++ Performance Measurements




(a) Normalized execution time of SPADE in C++ and OptiML with and without fusing optimizations, for 1 to 8 CPU cores. Speedup relative to 1-core OptiML with fusing on top of each bar.

(b) Normalized execution time of template matching in C++ and OptiML, for 1 to 8 CPU cores. Speedup numbers relative to 1-core OptiML shown on top of each bar.

(c) Normalized execution time of 8-core CPU and 1 GPU for a selection of applications in OptiML. Speedup numbers relative to 1-core CPU shown on top of each bar.

Figure 16.9: Performance of OptiML compared to C++

Figures 16.9(a) and 16.9(b) show how OptiML performs relative to hand-optimized C++ versions for two real-world applications: SPADE (discussed in Section 14.1) and a template matching application [9] used for object recognition in robotics. For each experiment, we ran each application 10 times and report the mean execution time of the final 5 executions (not counting initialization).

The results show that OptiML generates code that performs comparably to, and can even outperform, a hand-tuned C++ implementation. For SPADE, prior to applying the optimizations discussed in the previous section, the code generated by OptiML is three times slower than the C++ version. This overhead comes from the extra memory allocations and loops over the data that are required in the naive implementation. The naive version also does more computation than the C++, because it always initializes a StreamRow even when it is not needed. By applying fusion and code motion as we described, the OptiML code becomes a tight loop with no intermediate allocation. The small improvement over the C++ version might be due to the JVM JIT producing slightly better native code and from the removal of an atomic update, which is unnecessary on 64-bit platforms.

The OptiML version of the template mapping (TM) application is much shorter and performs better than the C++ version, mostly due to removing a substantial amount of low-level bit-manipulating optimizations from the application code that did not perform as well on our platform. The C++ code also reused some data structures in ways that made parallelizing non-trivial, while the OptiML code was implicitly parallel from extending Delite Ops. Because the C++ TM is sequential, we report only the single-threaded execution time; the OptiML version scales efficiently to multiple processors. This application clearly demonstrates that low-level implementation details in application code are a liability to performance portability.

Finally, Figure 16.9(c) shows our GPU performance for three selected machine learning applications: Riemann Boltzmann Machine (RBM), Naive Bayes (NB), and Gaussian Discriminant Analysis (GDA). The results shown are relative to single-threaded OptiML performance. These results show that a heterogeneous programming model is required to achieve best performance on these applications; neither the CPU nor the GPU is best for all cases. The OptiML code can run on either the CPU or GPU without any changes to the source code. In previous work, we compared OptiML performance to MATLAB across a wider range of applications, and found that it outperforms MATLAB's CPU performance and GPU performance in most cases [126].

# Chapter 17

# Other Projects

In addition to Delite, other projects have explored LMS and embedded compilers in slightly different ways.

## 17.1 StagedSAC

[1] Domain-specific languages (DSLs) can bridge the gap between high-level programming and efficient execution. However, implementing compiler tool-chains for performance oriented DSLs requires significant effort. Recent research has produced methodologies and frameworks that promise to reduce this development effort by enabling quick transition from library-only, purely embedded DSLs to optimizing compilation.

In this case study we report on our experience implementing a compiler for StagedSAC [142]. StagedSAC is a DSL for arithmetic processing with multidimensional arrays modeled after the stand-alone language SAC (*Single Assignment C* [114]). The main language feature of both SAC and StagedSAC is the *with-loop* construct that enables high-level and concise implementations of array algorithms. At the same time, the functional semantics of the two languages allow for advanced compiler optimizations and parallel code generation.

Using Lightweight Modular Staging (LMS) and Delite, we were quickly able to switch from a pure-library embedding to an optimizing compiler.We began our development by building a purely embedded DSL, which we call LibrarySAC. LibrarySAC is basically just a Scala library and, since it does not perform any optimization, is very slow. Lightweight modular staging allowed us to transform LibrarySAC into a complete compiler with only minor changes to the DSL syntax

StagedSAC was developed by Vlad Ureche, without previous knowledge of Scala, LMS and Delite, over the course of two semester projects. We can split StagedSAC's evolution into three phases:

- From scratch to the LibrarySAC implementation, which took approximately 2 months, including learning Scala and SAC

[1] *Credits:* Two doctoral school semester projects by Vlad Ureche [142]

- From LibrarySAC to the first StagedSAC prototype, which took approximately 2 months, including learning how to use the first versions of the LMS framework
- From the first StagedSAC prototype to the current StagedSAC compiler, which took approximately 3 months, including learning Delite

The same three phases are reflected in the obtained performance. Each stage of the evolution reduced the running time for the benchmarks by an order of magnitude.

## 17.2 Scala Integrated Query (SIQ)

[2] Interaction with relational databases can be troublesome for software developers. The relational data model does not match the data models of most general purpose programming languages. Embedding database queries as SQL Strings does not provide any compile time guarantees and can lead to run time exceptions or bugs like SQL injection vulnerabilities. This thesis introduces Scala Integrated Query (SIQ), a deep and type safe integration of database queries into Scala. It is similar to Microsoft LINQ and the LINQ-to-SQL provider on .NET, but provides stronger compile time guarantees, translates queries with nested results more efficiently and does not introduce new syntax. SIQ compiles a subset of Scala and a small relational library into SQL. Database tables, in-memory collections and tuples can be combined for arbitrarily nested results, unlike ScalaQuery, Squeryl or SQL, where results are always flat collections. Using the Ferry encoding allows translating nesting efficiently to one or more SQL queries, a number which only depends on the result type, unlike LINQ-to-SQL where it can depend on the size of involved tables. The prototype is implemented as a library without any special Scala compiler support. Several language features allow lifting queries to abstract syntax trees at run time, while providing precise static type safety. The thesis describes how the necessary constructs are lifted and how type safety can reflect the database schema and SQL specifics. The use of the Lightweight Modular Staging framework makes the library very modular and allows a high degree of code re-use when adapting type safety to different SQL dialects.

SIQ [151] compiles an embedded subset of Scala into SQL for execution in a DBMS. Advantages of SIQ over SQL are type safety, familiar syntax, and better performance for complex queries by avoiding avalanches of SQL queries.

## 17.3 Jet: High Performance Big Data Processing

[3] Cluster computing systems today impose a trade-off between generality, performance and productivity. Hadoop [157] and Dryad [64] force programmers to write low level programs that are efficient but tedious to compose. Systems like Dryad/LINQ [158] and Spark [159] allow concise modeling of user programs but do not apply relational optimizations. Pig [101] and

[2] *Credits:* Master's thesis by Christopher Vogt [151]
[3] *Credits:* Master's thesis by Stefan Ackermann [1]

Hive [138] restrict the language to achieve relational optimizations, making complex programs hard to express without user extensions. However, these extensions are cumbersome to write and disallow program optimizations.

We present a big data processing library called Jet. It uses Lightweight Modular Staging (LMS) and embedded compilation to analyze the structure of user programs and generate code for different cluster frameworks. Custom analyses are performed to apply early projection insertion, which eliminates unused data. Together with generic optimizations like code motion and loop fusion, these serve to highly optimize the performance-critical path of the system. The language embedding and a high-level interface make Jet programs expressive and resembling regular Scala code. Through its modular code generation scheme, Jet can execute the same programs on different cluster frameworks, including Spark and Hadoop. On Hadoop we achieve end-to-end speedups of up to 59% over straightforward implementations, while on Spark they can reach up to 149%.

## 17.4 JavaScript as an Embedded DSL

[4] Developing rich web applications requires mastering a heterogeneous environment: Although the server-side can be implemented in any language, on the client-side, the choice is limited to JavaScript. The trend towards alternative approaches to client-side programming (as embodied by CoffeeScript [4], Dart [50] & GWT [51]) shows the need for more options on the client-side. How do we bring advances in programming languages to client-side programming?

One challenge in developing a large code base in JavaScript is the lack of static typing, as types are helpful for maintenance, refactoring, and reasoning about correctness. Furthermore, there is a need for more abstraction and modularity. “Inversion of control” in asynchronous callback-driven programming leads to code with control structures that are difficult to reason about. A challenge is to introduce helpful abstractions without a big hit on performance and/or code size. Communication between the server side and the client side aggravates the impedance mismatch: in particular, data validation logic needs to be duplicated on the client-side for interactivity and on the server-side for security.

There are three widely known approaches for addressing the challenges outlined above. One is to create a standalone language or DSL that is compiled to JavaScript and provides different abstractions compared to JavaScript. Examples include WebDSL [150], Links [28, 29] and Dart [50]. However, this approach usually requires a lot effort in terms of language and compiler design, and tooling support, although WebDSL leverages Spoofax [71] to alleviate this effort. Furthermore, it is not always clear how these languages interact with other languages on the server-side or with the existing JavaScript ecosystem on the client-side.

Another approach is to start with an existing language like Java, Scala or Clojure and compile it to JavaScript. Examples include GWT [51], Scala+GWT [78] and Clojurescript [135]. This approach addresses the problem of impedance mismatch between client and server programming but comes with its own set of challenges. In particular, compiling Scala code to

[4] *Credits:* Master’s thesis by Grzegorz Kossakowski and doctoral school semester project by Nada Amin [79]

JavaScript requires compiling Scala's standard library to JavaScript as any non-trivial Scala program uses Scala collections. This leads to not taking full advantage of libraries and abstractions provided by the target platform which results in big code size and suboptimal performance of Scala applications compiled to JavaScript. For example, a map keyed by strings would be implemented natively in JavaScript as an object literal, while, in Scala, one would likely use the hash map from the standard library, causing it to be compiled to and emulated in JavaScript. Moreover, both approaches tend to not accommodate very well to different API design and programming styles seen in many existing JavaScript libraries.

A third approach is to design a language than is a thin layer on top of JavaScript but provides some new features. A prime example of this idea is CoffeeScript [4]. This approach makes it easy to integrate with existing JavaScript libraries but does not solve the impedance mismatch problem. In addition, it typically does not give rise to new abstractions addressing problems seen in callback-driven programming style, though some JavaScript libraries such as Flapjax [92] and Arrowlets [75] are specifically designed for this purpose.

We present a different approach, based on Lightweight Modular Staging (LMS) [110], that aims to incorporate good ideas from all the approaches presented above but at the same time tries to avoid their described shortcomings: Our approach is to embed JavaScript as an Embedded DSL in Scala [79].

### 17.4.1 Sharing Code between Client and Server

In addition to generating JavaScript / client-side code, we want to be able to re-use our DSL code on the Scala / server-side. In the LMS approach, the DSL uses the abstract type constructor `Rep` [93]. When generating JavaScript, this abstract type constructor is defined by IR nodes. Another definition, which we dub "trivial embedding", is to use the identity type constructor: `Rep[T] = T`. This is an example of the original polymorphic embedding idea [55, 15]. By stripping out `Rep`s in this way, our DSL can operate on concrete Scala types, replacing staging with direct evaluation. Even in the trivial embedding, when the DSL code operates on concrete Scala types, virtualization still occurs because the usage layer of the DSL is still in terms of abstract `Rep`s.

In the previous section, we showed how to define typed APIs to represent JavaScript external libraries or dependencies. In the trivial embedding, we need to give an interpretation to these APIs. For example, we can implement a Canvas context in Scala by using native graphics to draw on a desktop widget instead of a web page. We translated David Flanagan's Canvas example from the book "JavaScript: The Definitive Guide" [47], which draws Koch snowflakes on a canvas [48]. First, the translation from JavaScript to our DSL is straightforward: the code looks the same except for some minor declarations. Then, from our DSL code, we can generate JavaScript code to draw the snowflakes on a canvas as in the original code. In addition, via the trivial embedding, we can execute the DSL code in Scala to draw snowflakes on a desktop widget. Screenshot presenting snowflakes rendered in a browser using HTML5 Canvas and Java's 2D are presented in figure 17.1.

HTML5 Canvas is a standard that is not implemented by all browsers yet so a fall-back

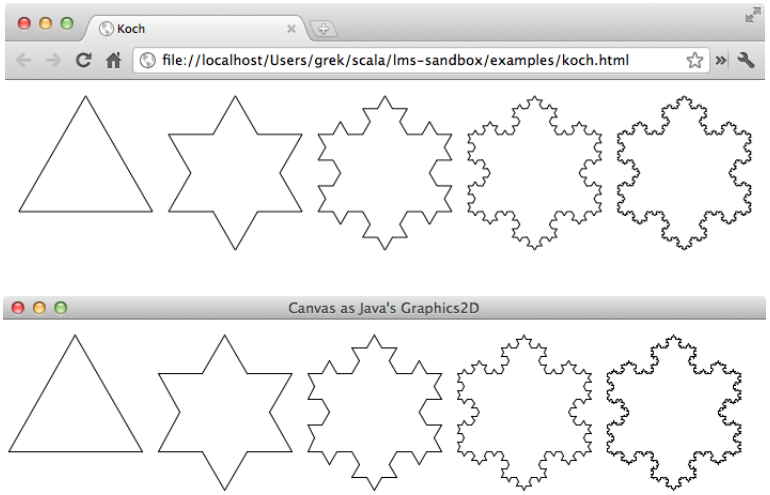



Figure 17.1: Snowflakes rendered using HTML5 Canvas and Java's 2D

mechanism is needed to support users of older browsers. This can be achieved through the trivial embedding by drawing using Java's 2D API, saving the result as an image and sending it to the browser. The decision to either generate a JavaScript snippet that draws on canvas and send it to the browser, or render the image on the server can be made at runtime (e.g. after inspecting information about the client's browser). In the case of rendering on the server-side, one can store computation that renders an image using Java's graphics 2D in a hash map and send back to the client the key as an url for an image. When a browser makes a second request, computation can be restored from the hash map, executed and the result sent back to the browser. All of that is possible because the computation itself is expressed against an abstract DSL API so we can swap implementations to either generate JavaScript or run the computation at the server side. Moreover, our DSL is just a library and computations are expressed as first-class values in a host language so they can be passed around the server program, stored in a hash map, etc.

### 17.4.2 Evaluation

We have implemented the embedded JavaScript DSL in Scala and developed a few web applications, which are simple but not trivial. In addition to drawing examples like the snowflakes of Figure 17.1, we extended the Twitter example from Section 12.3, which presents the latest tweets from selected users in an interactive way. In order to do so, we incorporated a useful subset of the DOM API and jQuery library using our typed APIs.

Finally, we developed a collaborative drawing application, which includes a server-side component (implemented using the Jetty web server). We use web sockets to communicate between the server and clients. Each client transmits drawing commands to the server, which broadcasts them to all the clients. When a new client joins, the server sends the complete drawing history to the new client, and the client reconstructs the image by playing the commands. A very simple improvement is to make the server execute the drawing commands as

well, and keep an up-to-date bitmap of the drawing – this can easily be achieved by using the trivial embedding described in Section 17.4.1. New clients then just obtain the bitmap instead of replaying the history, which can grow unboundedly.

# Chapter 18

# Related Work

This chapter briefly summarizes related work. Static program generation approaches include C++ templates [144], and Template Haskell [117]. Building on C++ templates, customizable generation approaches are possible through Expression Templates [146] e.g. used by Blitz++ [147]. An example of runtime code generation in C++ is the TaskGraph framework [7]. Active libraries were introduced by Veldhuizen [148], telescoping languages by Kennedy at al. [72]. Specific toolkits using domain-specific code generation and optimization include FFTW [49], SPIRAL [108] and ATLAS [156].

LMS draws inspiration from the work of Kiselyov et al. [76] on a staged FFT implementation and the work of Elliott et al. on compiling embedded DSLs [42]. Performing symbolic rewritings by defining operators on lifted expressions and performing common subexpression elimination on the fly is also central to their approach. LMS takes these ideas one step further by making them a central part of the staging framework itself.

Immediately related work on embedding typed languages includes that of Carette et al. [16] and Hofer et al. [55]. Lee et al. [83, 113] describe how LMS is used in the development of DSLs for high-performance parallel computing on heterogenous platforms.

Multi-Stage Programming Languages such as MetaML [133] and MetaOCaml [13] have been proposed as a disciplined approach to building code generators. These languages provide three syntactic annotations, *brackets*, *escape* and *run* which together provide a syntactic quasi-quotation facility that is similar to that found in Lisp but statically scoped and statically typed.

MSP languages make writing program generators easier and safer, but they inherit the essentially syntactic notion of combining program fragments. On the one hand, MSP languages transparently support staging of all language constructs, where LMS components have to be provided explicitly. On the other hand, the syntactic MSP approach incurs the risk of duplicating or reordering computation. [25, 129]. Code duplication can be avoided systematically by writing the generator in continuation-passing or monadic style, using appropriate combinators to insert `let`-bindings in strategic places. Often this is impractical since monadic style or CPS significantly complicates the generator code. The other suggested solution is to make extensive use of side-effects in the generator part, either in the form of mutable state or by using delimited control operators. However, uncontrolled side-effects during generation

pose safety challenges and invalidate much of the static guarantees of MSP languages. This dilemma is described as an "agonizing trade-off", due to which one "cannot achieve clarity, safety, and efficiency at the same time" [70].

By contrast, LMS prevents code duplication by handling the necessary side effects inside the staging primitives, which are semantic combinators instead of syntactic expanders. Therefore, code generators can usually be written in purely functional direct style and are much less likely to cause scope extrusion or invalidate safety assurances in other ways.

Another characteristic of some MSP languages is that staged code cannot be inspected due to safety considerations. This implies that domain-specific optimizations must happen before code generation. One approach is thus to first build an intermediate code representation, upon which symbolic computation is performed, and only then use the MSP primitives to generate code [76]. The burden of choosing and implementing a suitable intermediate representation is on the programmer. It is not clear how different representations can be combined or re-used. In the limit, programmers are tempted to use a representation that resembles classic abstract syntax trees (AST) since that is the most flexible. At that point, one could argue that the benefit of keeping the actual code representation hidden has been largely defeated.

Lightweight modular staging provides a systematic interface for adding symbolic rewritings. Safety is maintained by exposing the internal code structure only to rewriting modules but keeping it hidden from the client generator code.

The Delite compiler framework builds upon numerous previously published work in the areas of DSLs, staged metaprogramming, extensible compilation, and parallel programming.

The embedding of DSLs in a host language was first described by Hudak [58]. Tobin-Hochstadt et al. [139] extend Racket, a scheme dialect, with constructs to enhance the embeddability of other languages. We also needed to enhance the Scala compiler to allow for a deeper embedding of our DSLs. Feldspar [6] is an instance of an embedded DSL that combines shallow and deep embedding of domain operations to generate high performance code. Delite provides a framework that allows similar embeddings but can be re-used accross many DSLs. Telescoping languages [73] is a strategy to automatically generate optimized domain-specific libraries. Delite optimizes both the DSL and the program using it in one step. Elliot et al. [42, 84] pioneered embedded compilation and used simple image synthesis DSL as an example. Delite draws from that research extending it and applying it in other domains. Our approach also involved modifying the host language compiler to allow for a deeper embedding (e.g. we can lift conditionals and anonymous class constructors).

The ability to compile high-level languages to lower-level programming models has been investigated in a few contexts. Mainland et al. [87] use type-directed techniques to compile an embedded array language, Nikola, from Haskell to CUDA. This approach also suffers from the inability to overload some of Haskell's syntax (if-then-else expressions). Nystrom et al. [98] shows a library based approach to translating Scala programs to OpenCL code. This is largely achieved through java bytecode translation. A similar approach is used by Lime [5] to compile high-level Java code to a hardware description language such as Verilog.

Our compiler infrastructure implements a set of advanced compiler optimizations in a reusable fashion. Related work includes program transformations using advanced rewrite rules

[10], combining analyses and optimizations [85, 149, 23] as well as techniques for eliminating intermediate results [153, 30] and loop fusing [52].

Extensible compilers have been studied for a long time, recent examples in the Java world are Polyglot [97] and JastAdd [41]. The Glasgow Haskell Compiler also allows custom rewrite rules [67].

Existing parallel programming models operate at various different levels of abstraction. OpenCL [136] provides a standard that allows a programmer to target multiple different hardware devices from a single environment rather than using a distinct vendor API for each device. Higher level data-parallel programming models often provide implicit parallelization by providing the programmer with a data-parallel API that is transparently mapped to the underlying hardware. Recent work in this area includes Copperhead [17], which automatically generates and executes Cuda code on a GPU from a data-parallel subset of Python. Array Building Blocks [62] manages execution of data-parallel patterns across multiple processor cores and targets different hardware vector units from a single application source. DryadLINQ [63] converts LINQ [90] programs to execute using Dryad [64], which provides coarse-grained data-parallel execution over clusters. FlumeJava [20] targets Google's MapReduce [34] from a Java library and fuses operations in the data-flow graph in order to generate an efficient pipeline of MapReduce operations. Delite provides a framework for constructing new implicitly parallel DSLs and supports both implicit task and data parallelism.

# Bibliography


[1] Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. Jet: An Embedded DSL for High Performance Big Data Processing. In *International Workshop on End-to-end Management of Big Data*, Big Data, 2012.

[2] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. Fast partial evaluation of pattern matching in strings. *ACM Trans. Program. Lang. Syst.*, 28(4):696–714, 2006.

[3] Andrew W. Appel. SSA is Functional Programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[4] Jeremy Ashkenas. CoffeeScript. http://jashkenas.github.com/coffee-script/.

[5] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 89–108, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6.

[6] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The design and implementation of feldspar an embedded language for digital signal processing. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24275-5.

[7] Olav Beckmann, Alastair Houghton, Michael R. Mellor, and Paul H. J. Kelly. Runtime Code Generation in C++ as a Foundation for Domain-Specific Optimisation. In *Domain-Specific Program Generation*, pages 291–306, 2003.

[8] Geoffrey Belter, Elizabeth R. Jessup, Ian Karlin, and Jeremy G. Siek. Automating the generation of composed linear algebra kernels. In *SC*. ACM, 2009. ISBN 978-1-60558-744-8.

[9] G. Bradski and M. Muja. BiGG Detector. http://www.ros.org/wiki/bigg_detector, 2010.

[10] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundam. Inf.*, 69:123–178, July 2005. ISSN 0169-2968.

[11] Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *PACT*, pages 89–100, 2011.

[12] Rod M. Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *J. ACM*, 24(1):44–67, 1977.

[13] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *GPCE*, pages 57–76, 2003.

[14] Jacques Carette and Oleg Kiselyov. Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code. In *GPCE*, pages 256–274, 2005.

[15] Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally Tagless, Partially Evaluated. In Zhong Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 2007. ISBN 978-3-540-76636-0.

[16] Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5): 509–543, 2009.

[17] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. doi: http://doi.acm.org/10.1145/1941553.1941562.

[18] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, 2011.

[19] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *OOPSLA*, pages 835–847, 2010.

[20] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3.

[21] Cliff Click. Global Code Motion / Global Value Mumbering. In *PLDI*, pages 246–257, 1995.

[22] Cliff Click. Fixing the Inlining Problem. http://www.azulsystems.com/blog/cliff/2011-04-04-fixing-the-inlining-problem, 2011.

[23] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17:181–196, March 1995. ISSN 0164-0925.

[24] Cliff Click and Michael Paleczny. A Simple Graph-Based Intermediate Representation. In *Intermediate Representations Workshop*, pages 35–49, 1995.

[25] Albert Cohen, Sébastien Donadio, María Jesús Garzarán, Christoph Armin Herrmann, Oleg Kiselyov, and David A. Padua. In search of a program generator to implement generic transformations for high-performance computing. *Sci. Comput. Program.*, 62 (1):25–46, 2006.

[26] Charles Consel and Olivier Danvy. Partial Evaluation of Pattern Matching in Strings. *Inf. Process. Lett.*, 30(2):79–86, 1989.

[27] J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[28] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. In *Proceedings of the 5th international conference on Formal methods for components and objects*, FMCO'06, pages 266–296, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-74791-5, 978-3-540-74791-8. URL http://dl.acm.org/citation.cfm?id=1777707.1777724.

[29] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The Essence of Form Abstraction. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 205–220, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89329-5. doi: http://dx.doi.org/10.1007/978-3-540-89330-1_15. URL http://dx.doi.org/10.1007/978-3-540-89330-1_15.

[30] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP*, pages 315–326, 2007.

[31] Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *OOPSLA*, pages 341–360, 2010.

[32] Olivier Danvy and Andrzej Filinski. Abstracting Control. In *LISP and Functional Programming*, pages 151–160, 1990.

[33] Olivier Danvy and Andrzej Filinski. Representing Control: A Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[34] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[35] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[36] Iulian Dragos. *Compiling Scala for Performance*. PhD thesis, EPFL, Lausanne, 2010. URL http://library.epfl.ch/theses/?nr=4820.

[37] Gilles Dubochet. *Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming*. PhD thesis, EPFL, Lausanne, 2011. URL http://library.epfl.ch/theses/?nr=5007.

[38] Christopher Earl, Matthew Might, and David Van Horn. Pushdown Control-Flow Analysis of Higher-Order Programs. *CoRR*, abs/1007.4268, 2010.

[39] Jason Eckhardt, Roumen Kaiabachev, Emir Pasalic, Kedar N. Swadi, and Walid Taha. Implicitly Heterogeneous Multi-Stage Programming. *New Generation Comput.*, 25(3): 305–336, 2007.

[40] Jason L. Eckhardt, Roumen Kaiabachev, Kedar N. Swadi, Walid Taha, and Oleg Kiselyov. Practical Aspects of Multi-stage Programming. Technical Report TR05-451, Rice University, 2004.

[41] Torbjörn Ekman and Görel Hedin. The JastAdd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.

[42] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling Embedded Languages. In Walid Taha, editor, *SAIG*, volume 1924 of *Lecture Notes in Computer Science*, pages 9–27. Springer, 2000. ISBN 3-540-41054-6.

[43] Burak Emir, Martin Odersky, and John Williams. Matching Objects with Patterns. In *ECOOP*, pages 273–298, 2007.

[44] Andrzej Filinski. Representing Monads. In *POPL*, pages 446–457, 1994.

[45] Andrzej Filinski. Monads in action. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL*, pages 483–494. ACM, 2010. ISBN 978-1-60558-479-9.

[46] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *PLDI*, pages 237–247, 1993.

[47] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., 6th edition, 2011. ISBN 0596805527, 9780596805524.

[48] David Flanagan. koch.js. https://github.com/davidflanagan/javascript6_examples/blob/master/examples/21.06.koch.js, 2011.

[49] Matteo Frigo. A Fast Fourier Transform Compiler. In *PLDI*, pages 169–180, 1999.

[50] Google. Dart. http://http://www.dartlang.org/, .

[51] Google. GWT. http://code.google.com/webtoolkit/, .

[52] Clemens Grelck, Karsten Hinckfuß, and Sven-Bodo Scholz. With-Loop Fusion for Data Locality and Parallelism. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages*, volume 4015 of *Lecture Notes in Computer Science*, pages 178–195. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-69174-7.

[53] Miguel Guerrero, Edward Pizzi, Robert Rosenbaum, Kedar N. Swadi, and Walid Taha. Implementing DSLs in metaOCaml. In *OOPSLA Companion*, pages 41–42, 2004.

[54] Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410(2-3):202–220, 2009.

[55] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of dsls. In *GPCE*, pages 137–148, 2008.

[56] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, pages 267–276, 2011.

[57] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *ASPLOS*, 2012.

[58] Paul Hudak. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.*, 28 (4es):196, 1996.

[59] Paul Hudak. Modular Domain Specific Languages and Tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

[60] Jun Inoue and Walid Taha. Reasoning about Multi-stage Programs. In *ESOP*, pages 357–376, 2012.

[61] Intel. From a Few Cores to Many: A Tera-scale Computing Research Review. http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf, .

[62] Intel. Intel Array Building Blocks. http://software.intel.com/en-us/articles/intel-array-building-blocks, .

[63] Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 987–994, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2.

[64] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3.

[65] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-020249-5.

[66] Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FSTTCS*, volume 2 of *LIPIcs*, pages 383–414. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[67] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. Haskell, 2001.

[68] Ulrik Jørring and William L. Scherlis. Compilers and Staging Transformations. In *POPL*, pages 86–96, 1986.

[69] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung chieh Shan. Closing the stage: from staged code to typed closures. In Robert Glück and Oege de Moor, editors, *PEPM*, pages 147–157. ACM, 2008. ISBN 978-1-59593-977-7.

[70] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung chieh Shan. Shifting the stage: staging with delimited control. In *PEPM*, pages 111–120, 2009.

[71] Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA*, pages 444–463, 2010. doi: http://dx.doi.org/10.1145/1869459.1869497.

[72] Ken Kennedy, Bradley Broom, Keith D. Cooper, Jack Dongarra, Robert J. Fowler, Dennis Gannon, S. Lennart Johnsson, John M. Mellor-Crummey, and Linda Torczon. Telescoping Languages: A Strategy for Automatic Generation of Scientific Problem-Solving Systems from Annotated Libraries. *J. Parallel Distrib. Comput.*, 61(12):1803–1826, 2001.

[73] Ken Kennedy, Bradley Broom, Arun Chauhan, Rob Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John Mellor-Crummey. Telescoping Languages: A System for Automatic Generation of Domain Languages. *Proceedings of the IEEE*, 93(3):387–408, 2005. This provides a current overview of the entire Telescoping Languages Project.

[74] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred C. Chow. Partial redundancy elimination in SSA form. *ACM Trans. Program. Lang. Syst.*, 21(3): 627–676, 1999.

[75] Yit Phang Khoo, Michael Hicks, Jeffrey S. Foster, and Vibha Sazawal. Directing JavaScript with arrows. *SIGPLAN Not.*, 44:49–58, October 2009. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1837513.1640143. URL http://doi.acm.org/10.1145/1837513.1640143.

[76] Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In Giorgio C. Buttazzo, editor, *EMSOFT*, pages 249–258. ACM, 2004. ISBN 1-58113-860-1.

[77] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy Code Motion. In *PLDI*, pages 224–234, 1992.

[78] Grzegorz Kossakowski. ScalaGWT. http://scalagwt.github.com/.

[79] Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky. JavaScript as an Embedded DSL. In *ECOOP*, 2012.

[80] Shriram Krishnamurthi. *Linguistic reuse*. PhD thesis, Computer Science, Rice University, Houston, 2001.

[81] Peter J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[82] Julia L. Lawall and Peter Thiemann. Sound Specialization in the Presence of Computational Effects. In *TACS*, pages 165–190, 1997.

[83] HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *IEEE Micro*, 31(5):42–53, 2011.

[84] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122, 1999.

[85] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. *SIGPLAN Not.*, 37:270–282, January 2002. ISSN 0362-1340.

[86] Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 73–82, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-674-5.

[87] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 67–78, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. doi: http://doi.acm.org/10.1145/1863523.1863533. URL http://doi.acm.org/10.1145/1863523.1863533.

[88] MathWorks. MATLAB. http://www.mathworks.com/products/matlab/.

[89] John McCarthy. A Basis For A Mathematical Theory Of Computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963. doi: 10.1016/S0049-237X(08)72018-4.

[90] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 706–706, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0.

[91] Duane Merill, Michael Garland, and Andrew Grimshaw. High Performance and Scalable GPU Graph Traversal. Technical report, University of Virginia, 2011.

[92] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: http://doi.acm.org/10.1145/1640089.1640091. URL http://doi.acm.org/10.1145/1640089.1640091.

[93] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In Gail E. Harris, editor, *OOPSLA*, pages 423–438. ACM, 2008. ISBN 978-1-60558-215-3.

[94] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser Combinators in Scala. Technical Report CW491, Department of Computer Science, K.U. Leuven, 2008. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW491.abs.html.

[95] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. Scala-virtualized. In *PEPM*, pages 117–120, 2012.

[96] NVIDIA. CUDA. http://developer.nvidia.com/object/cuda.html.

[97] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *CC*, pages 138–152, 2003.

[98] Nathaniel Nystrom, Derek White, and Kishen Das. Firepile: run-time compilation for GPUs in scala. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 107–116, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0689-8.

[99] M. Odersky. Scala. http://www.scala-lang.org, 2011.

[100] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA*, pages 41–57, 2005.

[101] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[102] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999. URL http://ilpubs.stanford.edu:8090/422/. Previous number = SIDL-WP-1999-0120.

[103] Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *ICFP*, pages 218–229, 2002.

[104] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. *SIGPLAN Not.*, 41(9):50–61, 2006. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1160074.1159811.

[105] Frank Pfenning and Conal Elliott. Higher-Order Abstract Syntax. In *PLDI*, pages 199–208, 1988.

[106] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/361011.361073.

[107] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A Generic Parallel Collection Framework. In *Euro-Par (2)*, pages 136–147, 2011.

[108] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms. *IJHPCA*, 18(1):21–45, 2004.

[109] J.C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. 1975.

[110] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, pages 127–136, 2010.

[111] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[112] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, pages 317–328. ACM, 2009. ISBN 978-1-60558-332-7.

[113] Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Building-Blocks for Performance Oriented DSLs. In *DSL*, pages 93–117, 2011.

[114] Sven-Bodo Scholz. Single Assignment C: efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, 2003.

[115] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 341–352, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: http://doi.acm.org/10.1145/1596550.1596599.

[116] Amin Shali and William R. Cook. Hybrid partial evaluation. In *OOPSLA*, pages 375–390, 2011.

[117] Tim Sheard and Simon L. Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.

[118] Jeremy G. Siek. General purpose languages should be metalanguages. In John P. Gallagher and Janis Voigtländer, editors, *PEPM*, pages 3–4. ACM, 2010. ISBN 978-1-60558-727-1.

[119] Alexander V. Slesarenko. Polytypic Staging: a new approach to an implementation of Nested Data Parallelism in Scala. Technical Report 5, Keldysh Institute preprints, 2012, 2012. URL http://library.keldysh.ru/preprint.asp?id=2012-5&lg=e.

[120] Anthony M. Sloane. Lightweight Language Processing in Kiama. In *GTTSE*, pages 408–425, 2009.

[121] Brian Cantwell Smith. *Procedural reflection in programming languages*. PhD thesis, MIT, 1982.

[122] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC. In *ESOP*, pages 485–500, 1994.

[123] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A Positive Supercompiler. *J. Funct. Program.*, 6(6):811–838, 1996.

[124] Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Trans. Program. Lang. Syst.*, 22(2):224–264, 2000.

[125] G.L. Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

[126] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML*, pages 609–616, 2011.

[127] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Aleksandar Prokopec, Vojin Jovanovic, Philipp Haller, Manohar Jonnalagedda, Martin Odersky, and Kunle Olukotun. Scalable Development of Multiple High Performance Domain-Specific Languages. In *(unpublished manuscript)*, 2012.

[128] Eijiro Sumii and Naoki Kobayashi. A Hybrid Approach to Online and Offline Partial Evaluation. *Higher-Order and Symbolic Computation*, 14(2-3):101–142, 2001.

[129] Kedar N. Swadi, Walid Taha, Oleg Kiselyov, and Emir Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. In John Hatcliff and Frank Tip, editors, *PEPM*, pages 160–169. ACM, 2006. ISBN 1-59593-196-1.

[130] Walid Taha. *Multistage programming: its theory and applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Supervisor-Sheard, Tim.

[131] Walid Taha. A Sound Reduction Semantics for Untyped CBN Multi-stage Computation. Or, the Theory of MetaML is Non-trivial (Extended Abstract). In *PEPM*, pages 34–43, 2000.

[132] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *POPL*, pages 26–37, 2003.

[133] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[134] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-Stage Programming: Axiomatization and Type Safety. In *ICALP*, pages 918–929, 1998.

[135] The Clojure Team. ClojureScript. https://github.com/clojure/clojurescript/wiki.

[136] The Khronos Group. OpenCL 1.0. http://www.khronos.org/opencl/.

[137] Peter. Thiemann and Dirk Dussart. Partial evaluation for higher-order languages with state. Technical report, 1999. URL http://www.informatik.uni-freiburg.de/~thiemann/papers/mlpe.ps.gz.

[138] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[139] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 132–141, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8.

[140] Eric Torreborre. Specs: Software Specifications for Scala, 2011. URL http://etorreborre.github.com/specs2/.

[141] Valentin F. Turchin. The Concept of a Supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.

[142] Vlad Ureche, Tiark Rompf, Arvind K. Sujeeth, Hassan Chafi, and Martin Odersky. Staged-SAC: a case study in performance-oriented DSL development. In *PEPM*, pages 73–82, 2012.

[143] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/352029.352035.

[144] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002. ISBN 0201734842.

[145] Dimitrios Vardoulakis and Olin Shivers. CFA2: A Context-Free Approach to Control-Flow Analysis. In *ESOP '10*, pages 570–589, 2010. doi: 10.1007/978-3-642-11957-6_30.

[146] Todd Veldhuizen. Expression Templates. In Stanley B. Lippman, editor, *C++ Gems*, pages 475–488. SIGS Publications, Inc., New York, NY, 1996.

[147] Todd L. Veldhuizen. Arrays in Blitz++. In Denis Caromel, R. R. Oldehoeft, and Marydell Tholburn, editors, *ISCOPE*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer, 1998. ISBN 3-540-65387-2.

[148] Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, May 2004.

[149] Todd L. Veldhuizen and Jeremy G. Siek. Combining Optimizations, Combining Theories. Technical Report TR582, Indiana University, May 2003.

[150] Eelco Visser. WebDSL: A Case Study in Domain-Specific Language Engineering. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-88642-6. URL http://dx.doi.org/10.1007/978-3-540-88643-3_7.

[151] Jan Christopher Vogt. Type Safe Integration of Query Languages into Scala. Diplomarbeit, RWTH Aachen, Germany, 2011.

[152] Phil Wadler. The expression problem. Posted on the Java Genericity mailing list, 1998.

[153] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.

[154] Philip Wadler and Stephen Blott. How to Make ad-hoc Polymorphism Less ad-hoc. In *POPL*, pages 60–76, 1989.

[155] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java Multi-stage Programming Using Weak Separability. In *PLDI*, 2010.

[156] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[157] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, June 2009. ISBN 0596521979.

[158] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*, pages 1–14, 2008.

[159] Matei Zaharia. Spark. URL http://spark-project.org/.

[160] Matthias Zenger and Martin Odersky. Extensible Algebraic Datatypes with Defaults. In *ICFP*, pages 241–252, 2001.

# Curriculum Vitae

## Personal Information

| | |
|---|---|
| Name: | Tiark Rompf |
| Place of birth: | Bremen, Germany, 1981 |

## Education

| | |
|---|---|
| 2012 | PhD, Computer Science, EPFL |
| 2008 | MS ("Diplom"), Computer Science/Medical Informatics, Universität zu Lübeck |
| 2003 | BS ("Vordiplom"), Computer Science, Universität Bremen |
| 2000 | Abitur, Altes Gymnasium Bremen |

## Professional Experience

| | |
|---|---|
| 2008 | EPFL, research assistant |
| 2004 | R&S Medizinsysteme, co-founder |
| 2000 | Nachtlicht-Media, co-founder |
| 2000 | Bergmannsheil hospital Bochum, civil service |

## Awards

| | |
|---|---|
| 2011 | CACM research highlight |
| 2008 | EPFL PhD fellowship |
| 1997,2006 | Taekwondo full contact champion/vice-champion, federal state level |
| 1999,2001 | Jugend forscht (German youth research contest): 1st prize, federal state level |