Uniting Language Embeddings for Fast and Friendly DSLs

THÈSE N^{O} 6882 (2016)

PRÉSENTÉE LE 22 AVRIL 2016

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE MÉTHODES DE PROGRAMMATION 1
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Vojin JOVANOVIC

acceptée sur proposition du jury:

Prof. J. R. Larus, président du jury
Prof. M. Odersky, directeur de thèse
Prof. W. Taha, rapporteur
Dr S. Erdweg, rapporteur
Prof. C. Koch, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2016

To my son David.

Acknowledgements

Thank you Martin for being the best advisor. You taught me how to think hard, work hard, train hard, and at the same time be kind and generous. The past four years have been a rough ride for me. In all the tough moments I could always rely on your endless support and wise advice. Thank you for gathering the best team I worked in and for giving us all enough space to develop into great people.

Thank you Christoph for helping me out in critical moments, for amazing discussions, and great ideas. It was a great pleasure working with you on databases and programming languages. Thanks for gathering the DATA lab and all of its brilliant members.

I would like to thank Prof. Taha, Dr. Erdweg, and Prof. Larus for being members of my thesis committee. Thank you for your effort; it is a great honor to have you as my examiners.

Mano, Sandro, a big thank you for all the crazy discussions, paper deadlines, great jokes, pitchers of beer, and lovely voyages. Sandro, special thanks for being patient with my messy ideas and formulas. Amir, thanks for being the smartest student I had and for all the help and amazing discussions. Heather and co-Heather, you were always supportive and there for me. Sébastien you were an amazing office mate that has shown me what deep and thoughtful really means. Vlad, thanks for being my companion through the whole journey. Danielle your strong laughter has always made me feel happy. Special thanks to Aleksandar Prokopec for being my first contact in LAMP. Finally, I would like to thank Eugene Burmako, Denys Shablain, Hubert Plociniczak, Lukas Rytz, Adriaan Moors, Tiark Rompf, Nada Amin, George Nithin, Dimitry Petrasko, Ingo Meier, Fabien Salvi, Philipp Haller, Miguel Garcia, Vladimir Nikolaev, Nicolas Stucki, Tobias Schlatter, and Vera Salvisberg for all the great moments.

I would like to thank my closest friends Ivan and Ana for all the coolest moments and all the dawns we awaited together. Ivan thanks for all the sports we did, and the life hacks we discussed–they have helped me tremendously in the past years. I would also thank my cuddly friends Azra, Adrian, Maja, Petar, and the Petrovic family for making me feel like home in Lausanne. Thanks to the Tasev family for being my support through all these years. Finally, I would like to thank all my friends from Belgrade for being a crazy bunch and always inspiring me.

David, your infinitely cute smile and your big heart gave me the energy to fight to the

end. Although you are only four years old, you taught me much about life. I would never finish this journey without you by my side. Thank you my son!

I would not start nor finish this journey without the support from my lovely family. I would especially like to thank my mother—who is now driving a car next to me—for helping me in the most critical moments. My grandparents who, although they passed away, will be a pillars and great motivators for the rest of my life. And last but not least, my dad who thought me how to be intellectual, visionary, and enjoy life.

Darja, thank you for being the best mother for David and for taking great care of him. Thank you for inspiring me to start the PhD journey and supporting me to finish it. Without you, I would not be half the man I am today.

Lausanne, Switzerland, November 12th, 2015 V. J.

Abstract

The holy grail for a domain-specific language (DSL) is to be friendly and fast. A DSL should be friendly in the sense that it is easy to use by *DSL end-users*, and easy to develop by *DSL authors*. DSLs can be developed as entirely new compilers and ecosystems, which requires tremendous effort and often requires DSL authors to reinvent the wheel. Or, DSLs can be developed as libraries *embedded* in an existing host language, which requires significantly less effort.
Embedded DSLs (EDSLs) manifest a trade-off between being friendly and fast as they stand divided in two groups:

- *Deep* EDSLs trade user experience of both DSL authors and DSL end-users, for improved program performance. As proposed by Elliott et al., deep EDSLs build an *intermediate representation* (IR) of a program that can be used to drive domain-specific optimizations, which can significantly improve performance. However, this program IR introduces significant usability hurdles for both the DSL authors and DSL end-users. Correctly transforming programs is difficult and error prone for DSL authors, while error messages can be cryptic and confusing for DSL end-users.

- *Shallow* EDSLs trade program performance for good user experience. As proposed by P. Hudak, shallow EDSLs omit construction of the IR, i.e., they are executed directly in the host language. Although friendly to both DSL end-users and DSL authors, they can not perform domain-specific optimizations and thus exhibit inferior performance.

This thesis makes a stride towards achieving both (1) good user experience for DSL authors and DSL end-users, as well as (2) enabling domain-specific optimizations for improved performance. It unites shallow and deep DSLs by defining an *automatic translation* from end-user-friendly shallow DSLs, to better-performing deep DSLs. The translation uses reflection of the host language to cherry-pick the best of both shallow and deep EDSLs. During program development, a DSL end-user is presented with user friendly features of a shallow EDSL. Before execution in a production environment, programs are reliably translated into the high-performance deep EDSL with equivalent semantics.
Since maintaining both shallow and deep EDSLs is difficult, the thesis further shows how to reuse the shallow-to-deep translation to automatically generate deep EDSLs based on shallow EDSLs. With automatic generation of the deep EDSL, the DSL author is required

only to develop a shallow EDSL and the domain-specific optimizations for the deep EDSL that she deems useful. Finally, the thesis discusses a new programming abstraction that eases the development, of a specific kind, of deep EDSLs that are compiled in two stages. The new abstraction simplifies management of dynamic compilation in two-stage deep EDSLs.

Keywords: Embedded Domain-Specific Languages, Macros, Deep Embedding, Shallow Embedding, Compile-Time Meta-Programming, Dynamic Compilation

Résumé

Un langage de programmation dédié a trouvé le Graal s'il est facile à utiliser et rapide à la fois. Non seulement est-ce important pour un DSL d'être facile du point de vue d'un programmeur qui utilise le langage, mais aussi du point de vue du programmeur qui doit développé ce dernier. Soit nous développons un DSL en isolation : dans ce cas il faut implanter un compilateur et tout un écosytème d'outils spécifiques à ce langage, soit réinventer la roue ; soit nous l'implantons en tant que bibliothèque *embarquée* dans dans un langage général pré-existent. Cette dernière alternative demande moins d'effort de développement que la première.
Un langage embarqué, ou intégré (EDSL), introduit à son tour une opposition entre performance et expérience du programmeur. Nous pouvons distinguer deux types d'EDSLs :

- Une intégration approfondie (deep embedding, deep EDSL) privilégie la performance du programme à l'expérience des développeurs. Tel que proposé par Elliott et al., un "deep EDSL" utilise des représentation intermédiare du programme. Ces dernières peuvent être utilisées à des fins d'optimisations dédiées (connus du sous-domaine dans lequel nous opérons) ; ceci peut contribuer de manière significative à l'amélioration des performances. En contrepartie, la représentation intermédiate introduit des embûches difficiles à franchir, autant pour les auteurs de la EDSL que pour ses utilisateurs. Les premiers sont confrontés à des transformations de programmes difficiles, et les derniers sont confrontés à des messages d'erreurs cryptiques.

- Une intégration de surface (shallow embedding, shallow EDSL), tels que proposés par P. Hudak, omettent les représentations intermédiares, et sont donc plus faciles d'utilisation. Malheureusement elles ne peuvent faire des optimisations dédiées, et donc souffrent en terme de performance.

Dans cette thèse nous montrons qu'il est possible de réconcilier la performance et l'expérience de développement dans les DSLs embarqués. Ceci est possible grâce une *traduction automatique* d'une "shallow embedding" vers une "deep embedding". La traduction utilise la réflection du langage hôte afin de cueillir attentivement les parties les plus utiles de chaque type d'EDSL. Ainsi, un utilisateur fait face à une intégration de surface lors du développement. Lorsqu'il est question d'éxécuter le programme, ce dernier est traduit dans son équivalent sémantique "approfondi".

Abstract

Il n'est certes pas raisonnable de maintenir à la fois une intégration de surface et approfondie. Nous montrons dans cette thèse qu'en utilisant les mécanismes de traduction automatique, nous pouvons de plus *générer* une version approfondie à partir une version de surface. Ainsi un développeur d'EDSLs n'a besoin que d'implanter une bibliothèque simple, ainsi que de spécifier les optimisations dédiées qui lui semblent importantes. Finalement, nous proposons une nouvelle abstraction qui permet de simplifier encore plus le processus de développement d'un DSL embarqué. Cette abstraction simplifie la gestion de la compilation dynamique.

Mots clefs : Langages Dédiés Embarqués, Macros, Intégration Approfondie, Intégration de Surface, Metaprogrammation à la compilation, Compilation Dynamique

Contents

List of Figures

List of Tables

1 Introduction

Our society's infrastructure is controlled by program code executed in data-centers, mobile devices, routers, personal computers, and device controllers. That program code is mostly written in *general-purpose programming languages* such as JavaScript, Java, Python, and C#. Advancements in our society's infrastructure depend on advancements of the program code that controls it and evolution of program code is directly influenced by programmer productivity.

Modern general-purpose programming languages allow programmers to be productive by providing constructs that allow high levels of *abstraction*. Good, high-level abstractions lead to concise programs that are easy to comprehend. Unfortunately, abstraction comes with a cost: abstractions require many indirections that, when executed on a *target platform*, make programs inefficient.

Inefficient programs slow down decision making and use excess energy for computation. Inefficiencies increase the running time of programs and, thus, postpone decisions that depend on program results. Long running programs also consume more energy. The amount of energy used for computation is becoming a significant portion of the overall energy consumption in the world. It is estimated that 2% of electricity budget in the United States is used for only data center computations [Mukherjee et al., 2009]. If we would write more efficient programs the IT infrastructure would advance faster and consume less energy.

Writing efficient programs in general-purpose programming languages, however, leads us back to low productivity. To make programs efficient, programmers usually remove abstractions and hand-craft their programs for a particular platform where the program is executed [Lee et al., 2011]. The problem becomes even worse on *heterogeneous platforms* where programmers are faced with multiple *computing targets* such as parallel CPUs, GPUs, and FPGAs. With heterogeneous platforms programmers must specialize their programs for each target separately.

Why general-purpose languages can not optimize programs that use abstractions? For compilers of general-purpose programming languages it is hard to remove the abstraction overhead and at the same time target heterogeneous platforms. The main reasons for this are:

- General purpose compilers reason about general computations. They are agnostic to *specific domains* such as linear algebra and relational algebra. This reduces the number of possible optimizations they can perform.
- General purpose compilers are faced with an overwhelming number of choices for optimization. Each choice exponentially increases a *search space* that the compiler needs to explore. Finding a specific solution that is optimal for a given platform in this vast space is in most cases unfeasible. Having domain knowledge about operations (e.g., knowing operation costs) allows to more efficiently explore the search space and to guide the optimizer towards a close-to-optimal solution.
- Specific target platforms, such as GPUs, do not support code patterns that can be written in general-purpose programming languages. Once such code patterns are written it is hard or impossible for a compiler to transform them to executable code. If the code was written in a *restricted language* that allows only supported code patterns it would be much easier to target different platforms.

1.1 Domain-Specific Languages

Domain-specific languages (DSLs) provide a restricted interface that allows users to write programs at the high-level of abstraction and at the same time highly optimize them for execution on different target platforms. The restricted interface allows the optimizer to extract the *domain knowledge* from user programs. The domain knowledge is used to better define the space of possible program executions and to guide the optimizer in exploring that space. The restricted interface and the domain knowledge can further be used to target heterogeneous platforms [Chafi et al., 2010].

An example of a successful DSL is the Standard Query Language (SQL) that has millions of active users worldwide. SQL concisely expresses the domain of data querying and uses the knowledge about relational algebra to optimize data queries as good as performance experts. SQL, as such, provides the base for many enterprise applications in the world.

1.1.1 Kinds of DSLs

DSLs can be categorized in two major categories: *i)* as *external* DSLs that have a specialized compiler for the language, and *ii)* as *internal* or *embedded* DSLs that are embedded inside a general-purpose host language.

External DSLs. The implementation of a usable external (or *stand-alone*) DSL requires building a language and a *language ecosystem*. The language ecosystem consists of libraries and programs such as integrated development environments (IDE), debugging tools, code analysis tools, and documentation tools. Developing a language is a great undertaking that can, in many cases, outweigh the benefits of building an external DSL.

External DSLs usually start as concise restricted languages but through their development grow towards general-purpose languages. As DSLs become popular their language designers can not resist the user's demand for features of general-purpose languages. These features, as they are added after the initial language design, do not always fit well into the original language. For example, SQL in most databases supports constructs like loops, variables, and hash-maps which diverge from the domain of relational algebra.

Embedded DSLs. A promising alternative to external DSLs are *embedded DSLs* (EDSLs) [Hudak, 1996]. Embedded DSLs are hosted in a general-purpose language and reuse large parts of its ecosystem: *i)* IDE support, *ii)* tools (e.g., a debugger), *iii)* compilation pipeline (e.g., parser, type-checker, optimizations, and code generation), and *iv)* standard library. Since general-purpose languages are designed to support general purpose constructs, growth of DSLs towards general-purpose constructs is well supported.

For the purpose of the following discussion, we distinguish between two main types of embeddings: *shallow* and *deep* embeddings.

- In a shallow EDSL, values of the embedded language are *directly* represented by values in the host language. Consequently, terms in the host language that represent terms in the embedded language are evaluated directly into host-language values that represent DSL values. In other words, each step of evaluation in the host language is a step of evaluation in the embedded language.
- In a deep EDSL, values of the embedded language are represented *symbolically*, that is, by host-language data structures, which we refer to as the *intermediate representation (IR)*. Terms in the host language that represent terms in the embedded language are evaluated into this intermediate representation. An additional evaluation step is necessary to reduce the intermediate representation to a direct representation. This additional evaluation is typically achieved through *interpretation* of the IR in the host language, or through *code generation* followed by *execution*.

1.1.2 Comparison of DSL Kinds

In this section we will compare DSL kinds with respect to programmability and performance. As it is hard to quantify and exactly judge programmability, in the following

discussion we will classify whether a DSL kind is easy to program or not into three categories: *i)* good, *ii)* moderate, and *iii)* bad. Scientifically proving how programmable is a DSL kind would require user studies which we did not perform—we rather build on anecdotes.

To compare programmability of different DSL kinds we introduce two types of programmers:

- **DSL end-users** are people that use a DSL to model and solve their tasks. This is a larger group of programmers as, usually, there are more language users than language authors. Therefore, it is good to optimize the design of DSLs for this group of programmers.
- **DSL authors** are the programmers that develop domain-specific languages. This group is smaller than DSL end-users, but is still very important. If developing a DSL is hard, then it will be harder to introduce new DSLs and features of existing DSLs will be developed at a slower pace.

External DSLs. For the DSL end-users, it is, in the ideal case, easy to program in external DSLs. Given that the DSL authors implement a good language, the language syntax is crafted for the domain, and easy to comprehend and write. Error reporting and tooling should be built such that DSL end-users easily prevent, identify, and finally fix the errors in their programs.

For the DSL authors developing external DSLs is a big undertaking. Although, the development process is not hard as DSL authors design their own compiler, the amount of work required to build a language ecosystem is tremendous. Therefore, we categorize external DSLs as hard to develop.

Finally, external DSLs exhibit high performance. A language and its compiler can be designed such that they extract required domain-knowledge from user programs. This domain knowledge can then be used to optimize programs. Good example of high-performance DSLs is Spiral [Püschel et al., 2005], as it thoroughly defines and uses the domain knowledge to explore the entire search space in order to find optimal programs.

Shallow EDSLs. For the DSL end-users, it is easy to program in shallow DSLs but less so than the in external DSLs. Syntax and error reporting of the host language, typically, can not be modified to perfectly fit the domain. However, languages with flexible syntax [Moors et al., 2012, Odersky, 2010] and powerful type systems can closely model many domains. Some host languages have language extensions for introducing extensible syntax [Erdweg et al., 2011] and customizable error reporting [Hage and Heeren, 2007, Heeren et al., 2003] further improving the interface of DSLs. Finally,

shallow DSLs have perfect interoperability with the host language libraries as the values in the embedded language directly correspond to the values in the host language.

For the DSL authors, shallow EDSLs are easy to program, as their development is similar to development of host language libraries. This makes it easy to evolve the language and experiment with different language features. For DSLs with complex error reporting or extensible syntax, however, the development becomes more difficult for the DSL authors.

Shallowly embedded DSLs exhibit low performance. The lack of the intermediate representation prevents exploiting the domain knowledge to implement optimizations. Further, the embedding makes heavy use of host-language abstractions, which in turn lead to performance degradation. For relevant research on reducing the abstraction overheads in user program see [Futamura, 1999, Brady and Hammond, 2010, Taha and Sheard, 1997, Rompf and Odersky, 2012, Würthinger et al., 2013, Le Meur et al., 2004, Henglein and Mossin, 1994].

Deep EDSLs. For the DSL end-users deep EDSLs are not ideal and we argue that it is hard to program in them. The reification of the DSL IR inevitably leads to abstraction leaks (§4.2) such as convoluted interfaces, difficult debugging, incomprehensible type errors, run-time error reporting, and others (see 4.2).

For the DSL author developing a DSL that is deeply embedded is not easy. Unlike with external DSLs where the difficulty comes from the amount of work required to develop the language ecosystem, in deep embeddings it is difficult to introduce reification in the host language without compromising the interface. The DSL author is required to exploit complicated type system features to minimize the abstraction leaks caused by the deep embedding.

Deep EDSLs exhibit high performance. An important advantage of deep embeddings over shallow ones is that DSL terms can be easily manipulated by the host language. This enables domain-specific optimizations [Rompf and Odersky, 2012, Rompf et al., 2013b] that lead to orders-of-magnitude improvements in program performance, and multi-target code generation [Brown et al., 2011]. For certain languages, another advantage of deeply embedded DSLs is their compilation at host language run-time. Compilation at run-time allows for dynamic compilation [Auslander et al., 1996, Grant et al., 2000]: values in the host language are treated as constants during DSL compilation. Dynamic compilation can improve performance in certain types of DSLs (e.g., linear algebra and query languages). For languages that do not benefit from dynamic compilation the run-time compilation is only an overhead.

Comparison summary. Table 1.1 summarizes the previous discussion. We can see that none of the DSL kinds is ideal for both DSL end-users and DSL authors, and exhibits

high-performance. For this reason, depending on the domain that is being targeted by the language, some kinds of DSLs might be more suitable than the others.

Table 1.1 – Compares different DSL kinds with respect to DSL end-user friendliness, DSL author friendliness and performance.

	External DSLs	Shallow DSLs	Deep DSLs
For DSL end-users	good	good	bad
For DSL authors	bad	good	moderate
Performance	good	bad	good

Choosing the right DSL kind. Some DSLs greatly benefit from extracting the domain knowledge. Typically, languages for domains with well defined transformation rules (e.g., relational algebra, linear algebra, logical formulas, etc.) benefit the most, as those rules can be used to define the space of possible transformations. The DSL optimizer can explore the space of possible executions and find the optimal one.

For domains where programs can be transformed based on the domain knowledge, external and deeply embedded DSLs are a good fit. With those approaches the DSL author can extract the domain knowledge from programs and use it for optimizations. Some DSL authors choose to, use the deep embedding (e.g., OptiML [Sujeeth et al., 2011]), and some use external DSLs (e.g., WebDSL [Groenewegen et al., 2008]).

Shallow embeddings, on the other hand, are a good fit for languages where exploiting domain knowledge is not beneficial and where DSL end-users need features of general-purpose programming languages. Good examples of such DSLs are languages for generating content in formats like JSON and XML, testing frameworks, and Actors [Haller and Odersky, 2009].

1.2 Importance of Support for DSLs

To allow both high-level of abstraction and high performance of programs it is necessary enable wide adoption of domain-specific languages. However, from Table 1.1 we see that support for domain-specific languages is not ideal. External DSLs require tremendous amounts of work to be implemented, deep embeddings are not ideal in terms of DSL end-user experience and DSL author productivity, and shallow embeddings lack in ways to remove the abstraction overhead.

For wide adoption of domain specific languages it is imperative to improve support for DSLs. Support for DSLs should: *i)* improve experience for the DSL end-users in all kinds of DSLs, and *ii)* improve infrastructure for building external and deeply embedded domain specific languages.

In the recent years *language workbenches* [Fowler, 2005] such as Spoofax [Kats and Visser, 2010] and Rascal [Klint et al., 2009, van der Storm, 2011] have been designed to generate large parts of the language ecosystem based on a declarative specification. With language workbenches it is possible to generate the parser, type-checker, name binding logic [Konat et al., 2013], IDE support [Kats and Visser, 2010], and debuggers (for a detailed overview of language workbenches see work by Erdweg et al. [Erdweg et al., 2013]).

Support for deeply embedded domain-specific languages has improved the DSL end-user interface and ease of development. Frameworks like Forge [Sujeeth et al., 2013a], similarly to language workbenches, allow generating the deep embedding from a declarative specification. Scala-Virtualized [Rompf et al., 2013a] proposes overriding host language constructs to better support deeply embedded DSLs. Svenningsson and Axelsson [Svenningsson and Axelsson, 2013] propose combining shallow and deep embeddings for better user experience. For detailed comparison of these approaches see §12.

1.3 Uniting Shallow and Deep Embeddings

This dissertation discusses how to unite shallow and deep embeddings in order to improve user experience for embedded domain-specific languages. The thesis is divided in two parts: Part I discusses improving user experience in the deep embedding, and Part II describes automation of the deep embedding development in order to minimize the effort required by the DSL author.

1.3.1 Part 1: Improving User Experience in the Deep Embedding

We improve user experience in the deep EDSLs by exploiting the complementary nature of shallow and deep embeddings. We use the shallow embedding for program development when good DSL end-user interface is more important than performance. In production, when the DSL end-user interface is irrelevant and performance is important we use the deep embedding with equivalent semantics.

We define an automatic translation, based on *reflection*, from shallow programs into the corresponding deep programs (§5). The translation is configurable to support different types of deep embedding that we show in §6. Then we introduce DSL reification at host language compile-time (§7) to improve error reporting (§8) and reduce run-time overhead in the deep embedding (§9). Finally, we give a summary of the framework (§10).

The translation is similar to the translation of Carette et al. [Carette et al., 2009] except that: *i)* the translation is automated, and *ii)* the translation uses host language reflection to embed all aspects of the language, such as, error reporting and DSL compilation. The deep embedding that is used as a target is similar to the functors used by Oliver Danvy [Danvy, 1999] for partial evaluation. With the translation the deep embedding is

free to choose between higher-order abstract [Pfenning and Elliot, 1988] and abstract syntax trees as well as the design of the intermediate representation.

1.3.2 Part 2: Automating Development of the Deep Embedding

In the second part of the thesis we automate development of the deep embedding:

- By re-using the shallow-to-deep translation to automatically generate the deep embedding based on the shallow embedding (§13).
- By providing an abstraction in the deep embedding for tracking values that are used for dynamic compilation. The DSL compiler, then automatically manages dynamic compilation, by introducing compilation guards and code caches (§14).

1.4 Terminology

In §3 we introduce a term *direct embedding* for a particular kind of shallow embeddings. For embedded domain-specific languages we use the abbreviation EDSL, however, in cases where it is clear from the context we simply use DSL. For kinds of DSLs we interchangeably use terms: *i)* deep embedding and deep EDSL, *ii)* shallow embedding and shallow EDSL, and *iii)* direct embedding and direct EDSL. We also interchangeably use terms: *i)* DSL end-user, user, and programmer, and *ii)* DSL author and author.

For the programs whose execution results have influence on human or machine decisions and ultimately change the physical world, we use a colloquial term and say that programs run *in production*.

2 Background

In this chapter we provide a brief introduction to the Scala Programming Language [Odersky et al., 2014] (§2.1) and to deep EDSLs in Scala (§2.2). This chapter provides necessary background for comprehending this thesis. Throughout the thesis we assume that the reader has basic knowledge about programming languages.

2.1 The Scala Programming Language

Scala is a multi-paradigm programming language. It supports object-oriented programming as well as functional programming. The primary target of Scala is the Java Virtual Machine (JVM), although recently, Scala's dialects also target JavaScript.

2.1.1 Object-Oriented Features of Scala

Scala's types are organized in a *type hierarchy*. At the *top* of the hierarchy is the `Any` type with its two subtypes `AnyVal` and `AnyRef`. Type `AnyVal` is the supertype of all primitive types (e.g., `Int`) while `AnyRef` is the supertype of all *reference* types. At the bottom of the hierarchy, stands the type `Nothing` which can not be inhabited by values. Type `Null` is a subtype of all reference types and has a single instance `null`.

Types at the top of the hierarchy define *universal methods* that are available on all their subtypes and therefore all Scala types. An example of such method are methods `equals` and `hashCode` that are used for checking equality between two objects and computing a hash code of an object.

2.1.2 Functional Features of Scala

Besides methods Scala provides native support for functions. In Scala functions can be defined as terms with a special syntactic construct (e.g., `(x: Int)=> x + 1`). Functions

are internally represented as classes with an `apply` method that defines a function body. For each cardinality (number of parameters) of a function there is a corresponding Scala class. For example, function `(x: Int)=> x + 1` is represented with an anonymous subclass of `Function2`:

```
class anonymous$uid extends Function2[Int, Int] {
  def apply(x: Int) = x + 1
}
```

Functions and methods can be *curried*: they can have multiple *parameter lists*. A curried function is simply a function that returns another function. For methods, Scala has special syntax to support multiple parameters in the definition. The following example shows a curried method and a curried function:

```
def fill(v: Int)(size: Int) // curried method
(v: Int) => (size: Int) => fill(v)(size) // curried function
```

Scala function evaluation first executes function arguments and then the function body (*by-value* evaluation order). However, it is possible, at the method definition site, to declare method parameters as evaluated *by-name*: the function is evaluated before its arguments. To achieve by-name evaluation the type of a parameter must be perpended with `=>`. For example, a by-name parameter of type `Int` is written as `p: => Int`.

Scala supports *pattern matching* over terms. In Scala `case classes` are used to define that classes can be *deconstructed* with pattern matching. A case class is a data-type that, among other things, has a synthesized factory method `apply` and an *extractor* for deconstruction. Extractor is a method named `unapply` that is used by pattern matching to deconstruct an object. By using methods for deconstruction, Scala decouples the deconstruction of a type and its data representation [Emir et al., 2007].

The signature of an extractor method for a case class must correspond to the factory method that constructs the object. If the constructor is defined as `(T1,...,Tn) => U)` the deconstructor must have the signature `Any => Option[(T1,...,Tn)]`. Type `Option[_]` is an equivalent of the `Maybe` monad for Scala.

In deep DSLs it is common to perform deconstruction of internal nodes in order to transform them. In the following example we show a definition of the IR node that represents constants, and how we can use pattern matching on it:

```
case class Const[T](v: T) extends Exp
val node = Const(42) // factory method defined by the case class
node match {
  case Const(42) => true  // invokes a synthesized deconstructor
}
```

2.1.3 Implicit Parameters and Conversions

Value, object, and method definitions in Scala can be declared as *implicit*. Marking a definition as implicit allows the Scala compiler to use this method as an *implicit argument* or for *implicit conversions*. The definition is declared as implicit by writing a keyword implicit in front:

```
implicit val stream: PrintWriter
```

Methods in Scala can have implicit parameters. Only the parameters in the last parameter list of a curried method can be implicit. They are declared implicit by writing the keyword `implicit` in the beginning of the parameter list. Inside the method definition implicit parameters are further treated as implicit definitions. For example, a code generation method can accept an implicit `PrintWriter` by declaring it as implicit:

```
def emitNode(sym: Sym, rhs: Def)(implicit stream: PrintWriter) = {
// in the method body stream is treated as implicit
}
```

The implicit parameters can be passed explicitly by the programmer, however, if they are omitted, the Scala compiler tries to find an implicit definition that can satisfy that parameter. The `emitNode` method can be called in two ways:

```
emitNode(sym, rhs)(stream) // parameter passed explicitly
emitNode(sym, rhs)         // parameter added by the compiler
```

Type classes. Type classes are in Scala [Oliveira et al., 2010] introduced as a combination of traits, implicit definitions, and implicit parameters. A type class declaration is achieved by defining a trait with the interface for that type class, e.g., `Numeric[T]`. Then a type class instance is defined as an implicit definition that provides a type-class instance (i.e., instance of a trait) for a concrete type:

```
implicit val intNumeric: Numeric[Int] = new IntNumeric()
```

To *constrain* a type parameter of a method or a class one adds an implicit parameter that requires presence of a type-class instance. For example, constraining `T` to be `Numeric` is achieved by requiring an implicit instance of `Numeric` for type `T`:

```
def sum[T](xs: List[T])(implicit num: Numeric[T]): T
```

The same can be expressed with the shorthand notation for declaring type classes:

```
def sum[T: Numeric](xs: List[T]): T
```

Implicit conversions. Scala allows user-defined implicit conversions, besides the standard implicit conversions that are applied to primitive types. Implicit conversions are defined as implicit method definitions that have a single non-implicit parameter list and an optional implicit parameter list. An implicit conversion for an abstract type `Rep[Int]` can be defined as:

```
implicit def intOps(x: Rep[Int]): IntOps = new IntOps(x)
```

The implicit conversions are applied when a given term has an incorrect type. The compiler then tries to find an implicit conversion method that would "fix" the program to have correct types. Implicit conversions are categorized as: *i)* value conversions that happen when the *expected type* of term is not satisfied, and *ii)* method conversions that happen when the invoked method does not exist on a type. Given that `IntOps` has a method `+` the following example demonstrates both types of conversion:

```
val y: Rep[Int] = ...
val ops: IntOps = y // value conversion
y + 1               // method conversion
```

Extension methods. Implicit conversions subsume the mechanism of extension methods. In Scala an extension method is introduced by providing a wrapper-class (e.g., `IntOps`) that introduces the method and an implicit conversion that applies the wrapper. A class and an implicit conversion can be more concisely written as an `implicit class`. The following examples shows the implicit class that adds the `+` method to `Rep[Int]`:

```
implicit class IntOps(lhs: Rep[Int]) {
  def +(lhs: Rep[Int]): Rep[Int] = Plus(lhs, rhs)
}
```

2.1.4 Scala Macros

Scala Macros [Burmako, 2013] are a compile-time meta-programming feature of Scala. Macros operate on Scala abstract syntax trees (ASTs): they can construct new ASTs, or transform and analyze the existing Scala ASTs. Macro programs can use common functionality of the Scala compiler like error-reporting, type checking, transformations, traversals, and implicit search.

In this work we use a particular flavor of Scala macros called *black-box def macros*, though we will often drop the prefix "def" for the sake of brevity. From a programmer's point of view, def macros are invoked just like regular Scala methods. However, macro invocations are *expanded* during compile time to produce new ASTs. Macro invocations are type checked both before and after expansion to ensure that expansion preserves well-typedness. Macros have separated declarations and definitions: declarations are

represented to the user as regular methods while macro definitions operate on Scala ASTs. The arguments of macro method definitions are the type-checked ASTs of the macro arguments.

For DSLs in this thesis we use a macro that accepts a single block of code as its input. At compile time, this block is first type checked against the interface of the shallow embedding. We will use this type of macros for defining DSL boundaries, e.g., the following snippet is how we will define DSL programs:

```
vectorDSL {
  Vector.fill(1,3) + Vector.fill(2,3)
}
```

2.2 Deep Embedding of DSLs in Scala

The DSLs we show in this thesis are based on polymorphic embedding [Hofer et al., 2008] of DSLs in Scala. In the polymorphic embedding the DSLs are composed of Scala modules that contain their operations and types. In polymorphic embeddings the DSL types are represented as Scala's abstract types in two possible ways:

- As parametric types (similar to phantom types of Elliot et al.) `Rep[T]`. With this approach a type `T` in the host language is represented as `Rep[T]` and its semantics can be defined in multiple ways.
- As simple abstract types. In the embedded language a type `T` is represented as the abstract type `c.T` inside a component `c`.

Lightweight Modular Staging (LMS) is a staging [Taha and Sheard, 1997] framework and an embedded compiler for developing deeply embedded DSLs. LMS builds upon polymorphic embedding and provides a library of reusable language components organized as *traits* (Scala's first-class modules). An EDSL developer selects traits containing the desired language features, combines them through *mix-in* composition [Odersky and Zenger, 2005] and adds DSL-specific functionality to the resulting EDSL trait. EDSL programs then extend this trait, inheriting the selected LMS and EDSL language constructs.

Figure 2.1 illustrates this principle. The trait `VectorDSL` defines a simplified EDSL for creating and manipulating vectors over some numeric type `T`. Two LMS traits are mixed into the `VectorDSL` trait: the `Base` trait introduces core LMS constructs (e.g., abstract type `Rep`) and the `NumericOps` trait introduces the `Numeric` type class and the corresponding support for numeric operations. The bottom of the figure shows an example usage of the EDSL. The constant literals in the program are lifted to the IR, through *implicit conversions* introduced by `NumericOps`.

```
// The EDSL declaration
trait VectorDSL extends NumericOps with Base {
  object Vector {
    def fill[T:Numeric]
      (v: Rep[T], size: Rep[Int]): Rep[Vector[T]] =
      vector_fill(v, size)
  }

  implicit class VectorOps[T:Numeric]
    (v: Rep[Vector[T]]) {
    def +(that: Rep[Vector[T]]): Rep[Vector[T]] =
      vector_+(v, that)
  }
  // Operations vector_fill and vector_+ are elided
}

new VectorDSL { // EDSL program
  Vector.fill(1,3) + Vector.fill(2,3)
} // after execution returns a regular Scala Vector(3,6)
```

Figure 2.1 – Minimal EDSL for vector manipulation.

All types in the `VectorDSL` interface are instances of the parametric type `Rep[_]`. The `Rep[_]` type is an abstract type member of the `Base` LMS trait and abstracts over the concrete types of the IR nodes that represent DSL operations in the deep embedding. Its type parameter captures the type underlying the IR: EDSL terms of type `Rep[T]` evaluate to host language terms of type `T` during EDSL execution.

Operations on `Rep[T]` terms are added by implicit conversions (as extension methods) that are introduced in the EDSL scope. For example, the implicit class `VectorOps` introduces the `+` operation on every term of type `Rep[Vector[T]]`. In the example, the type class `Numeric` ensures that vectors contain only numerical values.

LMS has been successfully used by project Delite [Brown et al., 2011, Sujeeth et al., 2013b] for building DSLs that support heterogeneous parallel computing. EDSLs developed with Delite cover domains such as machine learning, graph processing, data mining, etc.

Part I
Improving User Experience with Deep Embeddings

3 Introduction: Concealing the Deep Embedding of DSLs

In §1 we introduced domain-specific languages and how they can be embedded into a host language. Then we discussed strengths and weaknesses of deep and shallow embeddings. In this section we compare deep and shallow embeddings and then show how they can be combined in order to keep all the strengths and cancel-out the weaknesses with embedded DSLs (EDSLs).

Deep EDSLs intrinsically *compromise programmer experience* by leaking their implementation details (§4.2). Often, IR construction is achieved through type system constructs that are, inevitably, visible in the EDSL interface. This can lead to cryptic type errors that are often incomprehensible to DSL end-users. In addition, the IR complicates program debugging as programmers cannot easily relate their programs to the code that is finally executed. Finally, the host language often provides more constructs than the embedded language and the usage of these constructs can be undesired in the DSL (§4.2.4). If these constructs are generic in type (e.g., list comprehensions or `try\catch`) they can not be restricted in the embedded language by using complex types (§4.2).

Shallow embeddings typically suffer less from *linguistic mismatch* than deep embeddings: this is particularly obvious for a class of shallow embeddings that we refer to as *direct* embeddings. Direct embeddings preserve the intrinsic constructs of the host language "on the nose". That is, DSL constructs such as `if` statements, loops, or function literals, as well as primitive data types such as integers, floating-point numbers, or strings are represented directly by the corresponding constructs of the host language.

Ideally, we would like to complement the high performance of deep EDSLs, along with their capabilities for multi-target code generation, with the usability of their directly embedded counterparts. Reaching this goal turns out to be more challenging than one might expect: let us compare the interfaces of a direct embedding and a deep embedding of a simple EDSL for manipulating vectors[1]. The direct version of the interface is declared

[1] All code examples are written in *Scala*. Similar techniques can be applied in other statically typed languages. Cf. [Carette et al., 2009, Lokhorst, 2012, Svenningsson and Axelsson, 2013].

as:

```
trait Vector[T] {
  def map[U](fn: T => U): Vector[U]
}
```

The interface of the deep embedding, however, fundamentally differs in the types: while the (polymorphic) `map` operation in the direct embedding operates directly on values of some generic type `T`, the deep embedding must operate on whatever intermediate representations we chose for `T`. For our example, similarly to Eliot et al. [Elliott et al., 2003], we chose the abstract higher-kinded type `Rep[T]`, to represent values of type `T` in the deep embedding:

```
trait Vector[T] {
  def map[U](fn: Rep[T => U]): Rep[Vector[U]]
}
```

The difference in types is necessarily visible in the signature and thus inevitably leaks into user programs. The `Rep` types immediately raise questions: is `Rep[T => U]` a function type? How is it applied then? Why not `Rep[T] => Rep[U]`? We will further see in §4.2, this difference in types is at the heart of many of the inconveniences associated with deep embeddings such as convoluted type errors, execution overhead, and inability to restrict the host language constructs. How then, can we conceal this fundamental difference?

In Forge [Sujeeth et al., 2013a], Sujeeth et al. propose maintaining two parallel embeddings, shallow and deep, with a single interface equivalent to the deep embedding. In the shallow embedding, `Rep` is defined to be the identity on types, that is `Rep[T] = T`, effectively identifying IR types with their direct counterparts. As a result, shallowly embedded programs may be executed directly to allow for easy prototyping and debugging. In production, a simple "flip of a switch" enables the deep embedding. Unfortunately, artifacts of the deep embedding still leak to the user through the fundamentally "deeply typed" interface. We would like to preserve the idiomatic interface of the host language and completely conceal the deep embedding.

The central idea of this work is the use of *reflection* to convert programs written in an unmodified direct embedding into their deeply embedded counterparts. Since the fundamental difference between the interfaces of the two embeddings resides in their types, we employ a configurable *type translation* to map directly embedded types T to their deeply embedded counterparts $[\![T]\!]$. For our motivating example the type translation is simply:

$$
\begin{array}{lcll}
[\![T]\!] & = & T & \text{if } T \text{ is in type argument position,} \\
[\![T]\!] & = & \texttt{Rep[}T\texttt{]} & \text{otherwise.}
\end{array}
$$

In §5 we describe this translation among several others and discuss their trade-offs.

Together with a corresponding translation on terms, the type translation forms the core of Yin-Yang, a generic framework for DSL embedding, that uses Scala's macros [Burmako, 2013] to reliably translate direct EDSL programs into corresponding deep EDSL programs. The virtues of the direct embedding are used during program development when performance is not of importance; the translation is applied when performance is essential or alternative interpretations of a program are required (e.g., for hardware generation).

Once we "broke the ice" by using reflection it becomes simpler to further improve the deep embeddings. Yin-Yang enables *domain-specific error reporting* at host-language compile time by compiling a DSL program during host-language compilation. It *restricts* the embedded language by providing an additional verification step for producing comprehensible error messages, and reduces run-time overhead by compiling the deep programs at host language compile-time.

Yin-Yang contributes to the state of the art as follows:

- It completely conceals leaky abstractions of deep EDSLs from the users. The virtues of the direct embedding are used for prototyping, while the deep embedding enables high-performance in production. The translation preserves well-typedness and ensures that programs written in the direct embedding will always be correct in the deep embedding. The core translation is described in §5.
- It allows choosing different deep embedding back-ends with simple configuration changes. We discuss different deep embeddings supported by Yin-Yang in §6.
- It improves error reporting (§8) in the direct embedding by: *i)* allowing domain-specific error reporting at host language compile-time (§8.2) and *ii)* restricting host language features in the direct EDSL based on the supported features of the deep DSL (§8.1).
- It reduces the run-time overhead of the deep EDSL programs (§9). The deep embeddings reify their IR before execution at runtime and thus impose execution overhead. For DSLs which are fundamentally not staged Yin-Yang uses *compile-time reification* to compile DSLs at host language compile time. For staged DSLs (compiled at host language run-time) Yin-Yang avoids re-reification of programs by storing them into a cache.

We evaluate Yin-Yang by generating 3 deep EDSLs from their direct embedding, and providing interfaces for 2 existing EDSLs. The effects of concealing the deep embedding and reliability of the translation were evaluated on 21 programs (1284 LOC), from EDSLs OptiGraph [Sujeeth et al., 2013b] and OptiML [Sujeeth et al., 2011]. In all programs

combined the direct implementation obviates 101 type annotations that were necessary in the deep embedding.

We use Yin-Yang as to introduce a user-friendly frontend for the Slick DSL [Typesafe]. This case study shows that developing a front-end for existing DSLs requires little effort and that developing an API with Yin-Yang takes far less time than developing the deep embedding. The complete evaluation is presented in §11.

4 Motivation: Abstraction Leaks in the Deep Embedding

The main idea of this work is that EDSL users should program by using a direct embedding, while the corresponding deep embedding should be used for achieving high-performance in production. To motivate this idea we consider the direct embedding and the deep embedding of a simple EDSL for manipulating vectors. Here, we use Scala to show the problems with the deep embedding that apply to other statically typed programming languages (e.g., Haskell and OCaml). These languages achieve the embedding in different ways [Svenningsson and Axelsson, 2013, Lokhorst, 2012, Carette et al., 2009, Guerrero et al., 2004], but this is always reflected in the type signatures. In the context of Scala, there are additional problems with type inference and implicit conversions that we discuss in §4.2.7.

Figure 4.1 shows a simple direct EDSL for manipulating numerical vectors. Vectors are instances of a `Vector` class, and have only two operations: *i)* vector addition (the `+`), and *ii)* the higher-order `map` function which applies a function `f` to each element of the vector. The `Vector` object provides factory methods `fromSeq`, `range`, and `fill` for vector construction. Note that though the type of the elements in a vector is generic, we require it to be an instance of the `Numeric` type class.

For a programmer, this is an easy to use library. Not only can we write expressions such as `v1 + v2` for summing vectors (resembling mathematical notation), but we can also get meaningful type error messages. This EDSL is an idiomatic library in Scala and displayed type errors are comprehensible. Finally, in the direct embedding, all terms directly represent values from the embedded language and inspecting intermediate values with the debugger is straightforward.

The problem, however, is that the code written in such a direct embedding suffers from major performance issues [Rompf et al., 2013b]. For some intuition, consider the following code for adding 3 vectors: `v1 + v2 + v3`. Here, each `+` operation creates an intermediate `Vector` instance, uses the `zip` function, which itself creates an intermediate `Seq` instance, and calls a higher-order `map` function. The abstractions of the language

```
object Vector {
  def fromSeq[T: Numeric](seq: Seq[T]): Vector[T] =
    new Vector(seq)
  def fill[T: Numeric](v: T, size: Int): Vector[T] =
    fromSeq(Seq.fill(size)(v))
  def range(start: Int, end: Int): Vector[Int] =
    fromSeq(Seq.range(start, end))
}
class Vector[T: Numeric](val data: Seq[T]) {
  def map[S: Numeric](f: T => S): Vector[S] =
    Vector.fromSeq(data.map(x => f(x)))
  def +(that: Vector[T]): Vector[T] =
    Vector.fromSeq(data.zip(that.data))
      .map(x => x._1 + x._2))
}
```

Figure 4.1 – The interface of a direct EDSL for manipulating numerical vectors.

that allow us to write code with high-level of abstraction have a downfall in terms of performance. Consecutive vector summations would perform much better if they were implemented with a simple while loop.

4.1 The Deep Embedding

For the DSL from Figure 4.1, the overhead could be eliminated with optimizations like stream fusion [Coutts et al., 2007] and inlining, but to properly exploit domain knowledge, and to potentially target other platforms, one must introduce an intermediate representation of the EDSL program. The intermediate representation can be transformed according to the domain-specific rules (e.g., eliminating addition with a null vector) to improve performance beyond common compiler optimizations [Rompf et al., 2013b]. To this effect, we use the LMS framework and present the deep version of the EDSL for manipulating numerical vectors in Figure 4.2.

In the `VectorDSL` interface every method has an additional implicit parameter of type `SourceContext` and every generic type requires an additional `TypeTag` type class[1]. The `SourceContext` contains information about the current file name, line number, and character offset. `SourceContext`s are used for mapping generated code to the original program source. `TypeTag`s carry all information about the type of terms. They are used to propagate run-time type information through the EDSL compilation for optimizations and generating code for statically typed target languages. In the EDSL definitions the

[1] `SourceContext` and `TypeTag` are an example of how information about source positions and run-time type information can be propagated. A particular DSL can use other types, however in Scala, they would still be passed through implicit parameters.

`SourceContext` is rarely used explicitly (i.e., as an argument). It is provided "behind the scenes" by implicit definitions that are provided in the DSL.

4.2 Abstraction Leaks in the Deep Embedding

The during their execution the deep embedding programs first construct an intermediate representation of the program. This section discusses how this IR construction inevitably leaks to the users through convoluted interfaces (§4.2.1), how it makes debugging difficult (§4.2.2), how type errors can become incomprehensible (§4.2.3), how it is not possible to restrict certain host language constructs (§4.2.4), how domain-specific error reporting can only be achieved at run time (§4.2.5), how run-time compilation at host-language runtime creates execution overhead (§4.2.6), and what are the Scala specific problems in the deep embeddings (§4.2.7).

4.2.1 Convoluted Interfaces

Embedding DSLs, by using the type system constructs of the host language, inevitably affects the language interface. This section shows how the language interface is convoluted with EDSLs based on generic types and with EDSLs based on non-generic types. Further, it shows the problems that arise when functions are not embedded in the DSL, and how DSLs that generate code must introduce additional parameters to their function definitions.

DSLs based on generic types. EDSLs typically use generic types to represent host-language types. A good example is LMS, where a type `T` from the host language is represented with the generic type `Rep[T]`[2]. Let us take the signature of the `map` function on vectors:

```
def map[U](fn: Rep[T => U]): Rep[Vector[U]]
```

Here, the use of `Rep` types immediately raises questions to non-expert users. What is the difference between types `Rep[T => U]` and `Rep[T] => Rep[U]`? What is the difference between `Vector[Rep[Int]]` and `Rep[Vector[Int]]`? Is it possible to write `Rep[Vector[Rep[Int]]]` and what would it mean?

Answers to these questions require knowledge of programming language theory and should not be exposed to domain experts, who might not be expert programmers. The generic types, that raise this kind of questions, are inevitably displayed to the user through method signatures, code documentation, and code auto-completion.

[2] In the approach of Elliot et al. [Elliott et al., 2003] the generic type is called `Exp`.

```
trait VectorDSL extends Base {
  val Vector = new VectorOps(Const(Vector))

  implicit class VectorOps(o: Rep[Vector.type]) {
    def fromSeq[T:Numeric:TypeTag](seq: Rep[Seq[T]])
      (implicit sc: SourceContext): Rep[Vector[T]] =
      vector_fromSeq(seq)
    def fill[T:Numeric:TypeTag](value: Rep[T], size: Rep[Int])
      (implicit sc: SourceContext): Rep[Vector[T]] =
      vector_fill(value, size)
    def range(start: Rep[Int], end: Rep[Int])
      (implicit sc: SourceContext):Rep[Vector[Int]]=
      vector_range(start, end)
  }

  implicit class VectorRep[T:Numeric:TypeTag]
    (v: Rep[Vector[T]]) {
    def data(implicit sc: SourceContext): Rep[Seq[T]] =
      vector_data(v)
    def +(that: Rep[Vector[T]])
      (implicit sc: SourceContext):Rep[Vector[T]] =
      vector_plus(v, that)
    def map[S:Numeric:TypeTag](f: Rep[T] => Rep[S])
      (implicit sc: SourceContext): Rep[Vector[S]] =
      vector_map(v, f)
  }

  // Elided IR constructors of ‘map‘, ‘data‘, ‘fromSeq‘, and ‘range‘
  case class VectorFill[T:TypeTag](v: Rep[T], s: Rep[Int],
    sc: SourceContext)
  def vector_fill[T:Numeric:TypeTag](v: Rep[T], size: Rep[Int])
    (implicit sc: SourceContext): Rep[Vector[T]] =
    VectorFill(v, size, sc) // IR node construction

  case class VectorPlus[T:TypeTag](lhs: Rep[T], rhs: Rep[T],
    sc: SourceContext)
  def vector_plus[T:TypeTag](l: Rep[Vector[T]], r: Rep[Vector[T]])
    (implicit sc: SourceContext): Rep[Vector[T]] =
    VectorPlus(l, r, sc) // IR node construction
}
```

Figure 4.2 – A deep EDSL for manipulating numerical vectors based on LMS.

DSLs based on non-generic abstract types. DSLs can also use non-generic abstract types (e.g., `this.T`) to represent a host-language type `T` [Hofer et al., 2008]. With these DSLs, method signatures are not significantly changed as the abstract types textually resemble the host-language types. However, this type of DSLs introduces additional constraints: *i)* DSLs must be defined inside modules in order to use abstract types, and *ii)* type abstraction over function types is not possible in Scala and functions are always executed in the host language.

Since DSLs are defined inside modules, the DSL end-users are exposed to a non-idiomatic interface:

- As using functions defined in the host language is not possible, although the signature of DSL methods indicates it should be. For example, if we have an increment function `inc: Int => Int` defined in the host language, we can not apply it with an argument of the `Int` type defined in the embedded language. This would yield a compilation error, although the signatures seem compatible.
- As users are required to define their programs inside a module that inherits the DSL module. This is necessary in order to avoid writing prefixes for types.
- As the documentation of the DSL methods and types is presented inside the modules.

Evaluating functions in the host language. The choice whether functions are embedded in the DSL, or left in the host language, affects how recursive functions must be written. If functions are left in the host language (i.e., `T => U` is represented as `Rep[T] => Rep[U]`) the recursion must be treated specially. Consider an example of a simple recursive function:

```
def fact(n: Rep[Int]): Rep[Int] =
  if (n == 0) 1
  else n * fact(n - 1)
```

Here, the factorial function recurses infinitely and never terminates. To prevent infinite recursion LMS requires users to modify the implementation and signature of `fact`:

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

Passing source information and type information. In DSLs that use code generation, method signatures must be enriched with source code information for purposes of

debugging (`SourceContext`) and type information for generating right types (`TypeTag`). This information also leaks in the DSL interface. In Scala `TypeTag`s and `SourceContext`s are passed with implicit parameters. This makes the interface harder to understand as the user of the EDSL, who might not be an expert programmer, needs to understand concepts like `TypeTag` and `SourceContext`. A method `map` from the introductory example (§3) with source and type information has two additional implicit parameters.

Examples of convoluted interfaces. The convoluted interfaces are well presented in Figure 4.2. Here we see how all methods have additional implicit arguments of type `SourceContext`, how method definitions are placed in an `implicit class` instead on the type it self, and how it is necessary to have additional methods for constructing the intermediate representation.

A good example of convoluted interfaces are data querying DSLs such as Slick [Typesafe]. Interface of these DSLs must accept tuples of combined `Rep` types and regular types. To accomodate this additional modifications must be added to the interface. As an example we show the `map` function of Slick:

```
def map[F, G, T](f: E => F)
  (implicit shape: Shape[_ <: FlatShapeLevel, F, T, G])
  : Query[G, T, C]
```

4.2.2 Difficult Debugging

In the methods of the direct EDSL all terms directly represent values in the embedded language (there is no intermediate representation). This allows users to trivially use debugging tools to step through the terms and inspect the values of the embedded language.

With the deep EDSL, user programs in the reification phase only instantiate the IR nodes. In the classical debugging mode this leads to difficulties as: *i)* users inspecting variable values will be faced with IR nodes, *ii)* the control flow follows all branches in the host language constructs as they get reified, and *iii)* stepping into the DSL operations will only display reificaiton logic.

Debugging generated code or an interpreter is more difficult as users cannot relate the debugger position to the original line of code. The domain-specific and general purpose optimizations applied to the program will likely reorder instructions and rename variables.

The only way to achieve debugging is to make exact maps from the generated code to the deep program and implement a specialized debugger. The debugger must track exact maps between the source code and the generated code. This requires extra effort and

decreases DSL author productivity.

4.2.3 Convoluted and Incomprehensible Type Errors

The `Rep[_]` types leak to the user through type errors. Even for simple type errors the user is exposed to non-standard error messages. In certain cases (e.g., incorrect call to an overloaded function), the error messages can become hard to understand. To illustrate, we present a typical type error for invalid method invocation:

```
found   : Int(1)
required: Vector[Int]
     x + 1
         ^
```

In the deep embedding the corresponding type error contains `Rep` types and the `this` qualifier:

```
found   : Int(1)
required: this.Rep[this.Vector[Int]]
    (which expands to) this.Rep[vect.Vector[Int]]
    x + 1
        ^
```

This example represents one of the most common type errors.

The errors get more involved when artifacts of language virtualization leak to the user:

```
val x = HashMap[Int, String](1 -> "one", 2 -> "two")
x.keys()
```

yields an error message with `SourceContext` parameters:

```
error: not enough arguments for method keys: (implicit pos
: scala.reflect.SourceContext)Prog.this.Rep[Iterable[Int]].
Unspecified value parameter pos.
  x.keys()
       ^
```

4.2.4 Unrestricted Host Language Constructs

In the deep embedding all generic constructs of a host language can be used arbitrarily. For example, `scala.List.fill[T](count: Int, el: T)` can, for the argument `el`, accept both direct and deep terms. This is often undesirable as it can lead to code explosion and unexpected program behavior.

In the following example, assume that generic methods `fill` and `reduce` are not masked by the `VectorDSL` and belong only to the host language library. In this case, the invocation of `fill` and `reduce` performs meta-programming over the IR of the deep embedding:

```
new VectorDSL {
  List.fill(1000, Vector.fill(1000,1)).reduce(_+_)
}
```

Here, at DSL compilation time, the program creates a Scala list that contains a thousand IR nodes for the `Vector.fill` operation and performs a vector addition over them. Instead of producing a small IR the compilation result is a thousand IR nodes for vector addition. This is a typical case of code explosion that could not happen in the direct embedding which does not introduce an IR.

On the other hand, some operations can be completely ignored. In the next example, the `try/catch` block will be executed during EDSL compilation instead during DSL program execution:

```
new VectorDSL {
  try Vector.fill(1000, 1) / 0
  catch { case _ => Vector.fill(1000, 0) }
}
```

Here, the resulting program always throws a `DivisionByZero` exception.

4.2.5 Domain-Specific Error Reporting at Runtime

Domain-specific languages often provide additional program analysis and verification beyond the type system. DSLs perform verification if data-sources are present [McClure et al., 2005, Zaharia et al., 2012], operations are supported on a given platform [Typesafe], whether multi-dimensional array shapes are correct [Ureche et al., 2012], etc. Ideally, error reporting with the domain-specific analysis should be performed at host language compile-time in order to avoid run-time errors in production and to reduce the time between the error is introduced and detected.

Deep EDSLs do not support compile-time error reporting due to their execution at host language run time. As a consequence, users must execute the DSL body in order to perform domain-specific error detection. This can lead to accidental errors in production code, requires more time to find the errors as the error reporting requires running tests, and errors are not integrated into the host language error reporting environment.

4.2.6 Runtime Overheads of DSL Compilation

The deep EDSLs are compiled at run time when the code is executed. This compilation introduces overhead on the first execution of the program as well as subsequent executions. The first execution has a larger overhead as the DSL needs to be fully compiled, and the overhead of subsequent executions depends on the implementation of the deep embedding. Some deep embeddings recompile their programs every time [Rompf and Odersky, 2012, Typesafe] which can lead to significant execution overhead [Shaikhha and Odersky, 2013]. Others support guards for re-compilation which lower the overhead but introduce additional constructs in the user programs, further leaking deep embedding abstractions.

4.2.7 Abstraction Leaks in the Deep Embedding Specific to Scala

Scala specific features like *weak least upper bounds* [Odersky et al., 2014] and *type erasure* lead to further abstraction leaks in the deep embedding. In this section we discuss those and show how they affect the DSL end-user.

Weak least upper bounds. The Scala language introduces *weak type conformance* where primitive types can conform although they are not in a subtyping relation. According to the language specification [Odersky et al., 2014], a type T weakly conforms to U ($T <:_\omega U$) if T is a subtype of U ($T <: U$) or T precedes U in the following ordering:

$$\texttt{Byte} <:_\omega \texttt{Short} <:_\omega \texttt{Int}$$
$$\texttt{Char} <:_\omega \texttt{Int}$$
$$\texttt{Int} <:_\omega \texttt{Long} <:_\omega \texttt{Float} <:_\omega \texttt{Double}$$

The weak least upper bounds are computed with respect to weak conformance: *i)* when inferring the return type of conditional statements (e.g., `if` and pattern matching), and *ii)* in type inference of type arguments. For example, term `if (cond) 1 else 1.0d` has a type `Double`. Weak least upper bounds are computed before the numeric conversions so the `then` branch of the previous term is widened to `1.0d`.

In case of the deep embedding there are no weak least upper bounds applied as term types are never primitive types, since they represent the IR. This leads to inconsistent behavior with the host language. For example, type checking

```
val res = if(positive) 1 else -1.0
res * 42
```

leads to the following type error:

```
error: value * is not a member of Prog.this.Rep[AnyVal]
res * 42
```

```
      ^
```

This behavior in the deep embedding can not be improved with implicit conversions as the conditional term is type correct and implicit conversions are triggered only for incorrect terms.

Type erasure. Scala introduces *type erasure* [Odersky et al., 2014] of abstract types to the base reference type (`AnyRef`). As a result, overloaded methods with different argument types erase to the same signature. Having two methods in the same scope with the same signature is illegal and leads to compilation errors.

In the deep embedding all types are abstract and erase to the type `Object` on the JVM. Consequently methods that could be defined in the direct embedding erase to the same signature and become illegal. Defining methods

```
def from(json: Rep[String]): Rep[Vector[Double]]
def from(bytes: Rep[Array[Byte]]): Rep[Vector[Double]]
```

yields an compilation error. This code would otherwise be valid in the host language.

The DSL authors are left with two options: *i)* to rename one of the methods and diverge from the original design (e.g., rename `from` to `fromBytes`, or *ii)* to introduce additional implicit parameters which will disambiguate the two methods after erasure

```
def from(json: Rep[String]): Rep[Vector[Double]]
def from(bytes: Rep[Array[Byte]])
  (implicit p: Overloaded): Rep[Vector[Double]]
```

where implicit argument `Overloaded` is always present in scope. Adding implicit arguments further convolutes the interface and increases compilation times thus causing additional abstraction leaks.

5 Translation of Direct EDSL Programs

The purpose of the core Yin-Yang translation is to reliably and automatically make a transition from a direct EDSL program to its deeply embedded counterpart. The transition requires a translation for the following reasons: *i)* host language constructs such as `if` statements are strongly typed and accept only primitive types for some of their arguments (e.g., a condition has to be of type `Boolean`), *ii)* all types in the direct embedding need to be translated into their IR counterparts (e.g., `Int` to `Rep[Int]`), *iii)* the direct EDSL operations need to be mapped onto their deeply embedded counterparts, and *iv)* methods defined in the deep embedding require additional parameters, such as run-time type information and source positions. To address these inconsistencies we propose a straightforward solution: a type-directed program translation from direct to deep embeddings.

Since the translation is type-directed it requires reflection infrastructure with support for *type introspection* and *type transformation*. The translation is based on the idea of representing language constructs as method calls [Carette et al., 2009, Rompf et al., 2013a] and systematically intrinsifying direct DSL operations and types of the direct embedding to their deep counterparts [Carette et al., 2009]. The translation operates in two main steps:

Language virtualization converts host language intrinsics into function calls, which can then be evaluated to the appropriate IR values in the deep embedding.

EDSL intrinsification converts DSL intrinsics, such as types and operations, from the direct embedding into their deep counterparts.

Figure 5.1 shows the translation pipeline of Yin-Yang. Language virualization is covered in §5.1 and DSL intrinsification in §5.2. The whole translation is explained on a concrete example in §5.2.4. Finally we argue for correctness of the translation in §5.2.5.

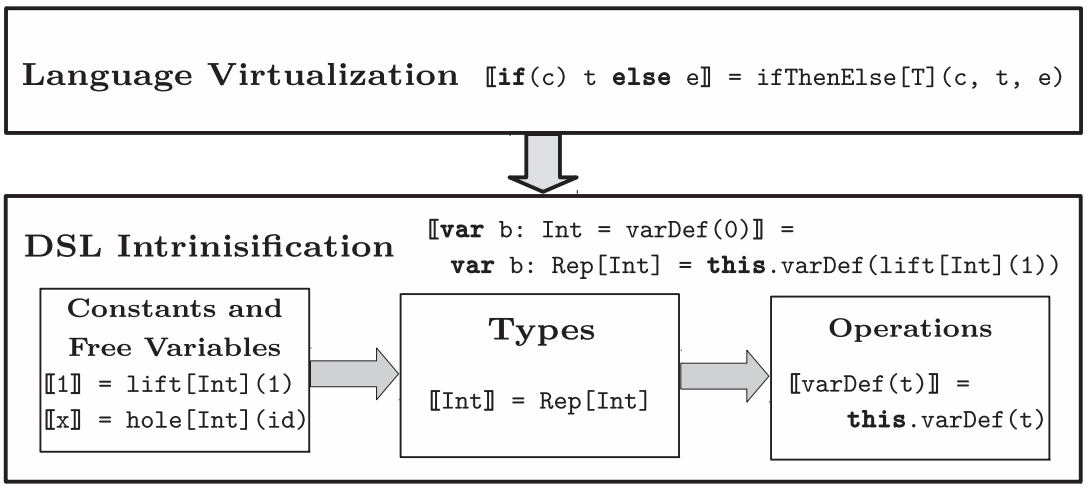


Figure 5.1 – Translation from the direct to the deep embedding.

5.1 Language Virtualization

Language virtualization allows to redefine intrinsic constructs of the host language, such as `if` and `while` statements. This can be achieved by translating them into suitable method invocations as shown by of Carette et al. [Carette et al., 2009] and Rompf et al. in the modified Scala compiler named Scala-Virtualized [Rompf et al., 2013a].

Yin-Yang follows the ideas of Scala-Virtualized but virtualizes all Scala language constructs that appear as expressions. Compared to Scala-Virtualized we translate additional language constructs: function/method definition and function application, exception handling, and all kinds of value-binding constructs (i.e., values, lazy values, and variables). Translation rules for supported language constructs are represented in Figure 5.2 with $[\![t]\!]$ denoting the translation of a term t. In some expressions the original types are introspected and used as a type argument of the corresponding virtualized method. These generic types are later translated to the deep embedding during the DSL intrinsification phase. Further Yin-Yang uses macros and reflection of unmodified Scala to achieve virtualization.

Defined translation rules convert language constructs into method calls where each language construct has a corresponding method. The signature of each method is partially defined by Yin-Yang. Method names, the number of type parameters and the number of type arguments are predefined while types of arguments and return types are open for the DSL author to define (§6).

Binding of the translated language constructs to the corresponding methods in the deep embedding is achieved during DSL intrinsification; language virtualization is agnostic of this binding. Further, in the implementation, all method names are prepended with $ [1]

[1] In Scala it is a convention that user defined method's names should not contain $ characters as those

Function Virtualization

$$\frac{\Gamma \vdash t : T_2}{[\![x : T_1 \Rightarrow t]\!] = \texttt{lam}[T_1, T_2](x : T_1 \Rightarrow [\![t]\!])} \qquad \frac{\Gamma \vdash t_1 : \ T_1 \Rightarrow T_2 \quad t_2 : \ T_1}{[\![t_1(t_2)]\!] = \texttt{app}[T_1, \ T_2]([\![t_1]\!], \ [\![t_2]\!])}$$

Method Virtualization

$$[\![\textbf{def } f[T_1](x : T_2) : T_3 = t]\!] = \textbf{def } f[T_1] : (T_2 \Rightarrow T_3) = [\![x : T_2 \Rightarrow t]\!]$$

$$\frac{\Gamma \vdash t_1.f : [T_1](T_2 \Rightarrow T_3)}{[\![t_1.f[T_1](t_2)]\!] = \texttt{app}[T_2, T_3]([\![t_1]\!].f[T_1], [\![t_2]\!])}$$

Control Constructs

$$\frac{\Gamma \vdash \textbf{if}(t_1) \ t_2 \ \textbf{else} \ t_3 : T}{[\![\textbf{if}(t_1) \ t_2 \ \textbf{else} \ t_3]\!] = \texttt{ifThenElse}[T]([\![t_1]\!], [\![t_2]\!], [\![t_3]\!])}$$

$$\frac{\Gamma \vdash \textbf{try } t_1 \ \textbf{catch } t_2 \ \textbf{finally } t_3 : \ T}{[\![\textbf{try } t_1 \ \textbf{catch } t_2 \ \textbf{finally } t_3 \ ]\!] = \texttt{try}[T]([\![t_1]\!], \ [\![t_2]\!], \ [\![t_3]\!])}$$

$$[\![\textbf{while}(c) \ b]\!] = \texttt{whileDo}([\![c]\!], \ [\![b]\!]) \qquad [\![\textbf{do } b \ \textbf{while}(c)]\!] = \texttt{doWhile}([\![c]\!], \ [\![b]\!])$$

$$[\![\textbf{return } t]\!] = \texttt{ret}([\![t]\!])$$

Value Bindings

$$[\![\textbf{val } x : T = t]\!] = \textbf{val } x : T = \texttt{valDef}[T]([\![t]\!])$$

$$[\![\textbf{lazy val } x : T = t]\!] = \textbf{val } x : T = \texttt{lazyValDef}[T]([\![t]\!])$$

$$[\![\textbf{var } x : T = t]\!] = \textbf{val } x : T = \texttt{varDef}[T]([\![t]\!])$$

$$\frac{\Gamma \vdash x : T}{[\![x]\!] = \texttt{read}[T]([\![x]\!])} \qquad \frac{\Gamma \vdash x : T}{[\![x = t]\!] = \texttt{assign}[T](x, [\![t]\!])}$$

Figure 5.2 – Rules for virtualization of Scala language intrinsics.

which avoids collisions with other user functions.

are reserved for the name mangling performed by the Scala compiler.

Functions. We virtualize function definition and application to support full abstraction over the host language expressions. This allows DSL authors to define how functions are treated by reifying them and optionally providing analysis and transformations over them. For example, DSL authors can define different inlining strategies, perform call graph analysis, or instrument all function calls. With Scala-Virtualized this is not possible as functions are not translated and thus it is impossible to abstract over them.

Methods. Method definitions follow a similar philosophy as functions. The difference is that in Scala, the `def` keyword is used to define universal quantification and/or recursion. This is similar to the `let` and `letrec` constructs in other functional languages. This translation is optional as, in some DSLs, it is more concise to reuse method application of the host language.

Control constructs. We translate all Scala control constructs (e.g., `if` and `try` to method calls. Scala's type system supports parametric polymorphism, by-name parameters, and partial functions that can model the semantics of all control constructs. How these features are used to model the original constructs is presented in §6).

Value bindings. Scala has multiple constructs for value binding: values, variables, and lazy values. Yin-Yang translates definition of all values into methods as well as value accesses. Abstraction over value access is necessary for tracking effects in case of variables, access order in case of lazy values, and for instrumentation and verification[2] in case of simple values.

Universal methods. Scala is designed such that the types `Any` and `AnyRef`, which reside at the top of the Scala class hierarchy, contain `final` methods. Through inheritance, these methods are defined on all types making it impossible to override their functionality without translation.

Yin-Yang virtualizes all methods on types `Any` and `AnyRef`. Method calls on objects are translated into the representation where the `this` pointer is passed as the first argument and, by convention, all methods start with a prefix `infix_`.

This representation is convenient for methods that are defined once for the whole hierarchy as the DSL author needs to define this method only once, as opposed to adding it to each data type. The caveat with this approach is that in case of methods that are overridden the virtual dispatch must be performed manually by the DSL author.

[2]Spores [Miller et al., 2014] by Miller et al. is an example where simple values should be virtualized for verification purposes.

For DSLs that require extension of these methods we provide an alternative translation of universal methods into the name mangled infix form:

$$[\![t_1 \texttt{ == } t_2]\!] = [\![t_1]\!] \texttt{ . \$ ==(} [\![t_2]\!] \texttt{)}$$

The `new` construct. The `new` construct of Scala is not virtualized to a single method. Signatures of data-type constructors differ in the number of type arguments, the number of arguments and in their types. If we used a single name for `new` virtualization, all data types would be constructed with a single method name that returns different result types based on the parameters. In Scala this would be possible by using a combination of overloading and implicit parameters, but this requires usage of complicated constructs in the deep embedding and we avoid it.

Instead, we rely on the translation and we virtualize constructor calls to method calls whose name depends on the type that is being constructed:

$$\frac{\Gamma \vdash \mathit{methodName} = \text{"}\texttt{new\_}\text{"} + \texttt{path}(\mathit{type})}{[\![\textbf{new } \mathit{type}[T](\mathit{arg})]\!] = \mathit{methodName}[T]([\![\mathit{arg}]\!])}$$

Methods on the `Any` type

$$[\![t_1 \texttt{ == } t_2]\!] = \texttt{infix\_ ==}([\![t_1]\!], [\![t_2]\!]) \qquad [\![t_1 \texttt{ != } t_2]\!] = \texttt{infix\_ !=}([\![t_1]\!], [\![t_2]\!])$$

$$[\![t.\#\#]\!] = \texttt{infix\_\#\#}([\![t]\!]) \qquad [\![t.\texttt{getClass}]\!] = \texttt{infix\_getClass}([\![t]\!])$$

$$[\![t.\texttt{isInstanceOf[T]}]\!] = \texttt{infix\_isInstanceOf[T]}([\![t]\!])$$

$$[\![t.\texttt{asInstanceOf[T]}]\!] = \texttt{infix\_asInstanceOf[T]}([\![t]\!])$$

Methods on the `AnyRef` type

$$[\![t_1 \texttt{ eq } t_2]\!] = \texttt{infix\_eq}([\![t_1]\!], [\![t_2]\!]) \qquad [\![t_1 \texttt{ ne } t_2]\!] = \texttt{infix\_ne}([\![t_1]\!], [\![t_2]\!])$$

$$[\![t.\texttt{notify}]\!] = \texttt{infix\_notify}([\![t]\!]) \qquad [\![t.\texttt{notifyAll}]\!] = \texttt{infix\_notifyAll}([\![t]\!])$$

$$[\![t.\texttt{wait}]\!] = \texttt{infix\_wait}([\![t]\!]) \qquad [\![t_1.\texttt{wait}(t_2)]\!] = \texttt{infix\_wait}([\![t_1]\!], [\![t_2]\!])$$

$$[\![t_1.\texttt{wait}(t_2, t_3)]\!] = \texttt{infix\_wait}([\![t_1]\!], [\![t_2]\!], [\![t_3]\!])$$

$$[\![t_1.\texttt{synchronized}[T](t_2)]\!] = \texttt{infix\_synchronized}[T]([\![t_1]\!], [\![t_2]\!])$$

Figure 5.3 – Rules for virtualization of methods on `Any` and `AnyRef`.

Not virtualizing class definitions. Yin-Yang does not virtualize class and trait definitions, including the *case class* definitions. This limitation, however, does not preclude class virtualization for embedded DSLs. We allow extensions to Yin-Yang that virtualize classes and traits through the use of the reflection API. The drawback of this approach is that DSL authors are required to know the reflection API which is more complex than the simple interface of language virtualization. For now, each framework that requires class virtualization Yin-Yang defines its own translation scheme.

Configuring method virtualization. When we virtualize methods and their application, we effectively override all expressions of Scala. In this case the DSL author has to: *i)* write the DSL definition in a way that corresponds to the translation and is not idiomatic to Scala, and *ii)* do additional transformation that removes IR nodes for function applications over domain-specific operations.

This can be cumbersome and we leave method virtualization as a configuration option that is disabled by default. By doing it, DSL authors can write DSLs in the Scala idiomatic way and the `app/lam` pairs for DSL operations never appear in the DSL intermediate representation.

5.1.1 Virtualizing Pattern Matching

The Scala compiler allows virtualization of pattern matching so its semantics can be overloaded. Yin-Yang reuses functionality of this pattern matcher in combination with DSL intrinsification to allow reasoning about pattern matching in DSL compilers. In this section, for purposes of explaining DSL intrinsification, we explain the functioning of the virtualized pattern matcher.

Scala's pattern matching can be interpreted with deconstructors and operations on the `Option` type of Scala. The successful match is represented with the `Some` type which is the monadic return of the `Option` monad and failures with the `None` type which is a monadic *zero* operation. Pattern nesting is represented with the monadic *bind* operation `flatMap`; alternation is represented with the `orElse` combinator on the `Option` monad which represents monadic addition. The semantics of Scala pattern matching can be completely represented with the operations of the zero-plus monad.

The virtual pattern matcher converts Scala pattern matching to the operations on a user-defined zero-plus monad. A pattern match is virtualized if the object `__match` is defined in scope. For the original semantics of Scala pattern matching is defined by the object `__match` defined in Figure 5.4. To virtualize pattern matching users can provide their own definitions of methods in `__match`, implementations of the zero-plus monad, and implementations of the case class deconstructors.

```
object __match {
  def zero: Option[Nothing] = None
  def one[T](x: T): Option[T] = Some(x)
  def guard[T](cond: Boolean, then: => T): Option[T] =
    if(cond) one(then) else zero
  def runOrElse[T, U](x: T)(f: T => Option[U]): U =
    f(x) getOrElse (throw new MatchError(x))
}
```

Figure 5.4 – Implementation of the virtualized pattern matcher with the semantics of Scala pattern matching and with `Option` as the zero-plus monad.

If the object `__match` is in scope, a simple pattern match

```
p match {
  case Pair(l, r) => f(l,r)
}
```

is translated into

```
__match.runOrElse(p) { x1: Any =>
  Pair.unapply(x1).flatMap(x2: (Int, Int) => {
    val l: Int = x2._1; val r: Int = x2._2;
    __match.one(f(l, r))
  })
}
```

In case of multiple case clauses

```
p match {
  case Pair(l, r) => f(l,r)
  case Tuple2(l, r) => f(l,r)
}
```

the monadic addition `orElse` is used for matching alternative statements in order:

```
Pair.unapply(p).flatMap(x2: (Int, Int) => {
  val l: Int = x2._1; val r: Int = x2._2;
  __match.one(f(l, r))
}).orElse(
  Tuple2.unapply(p).flatMap(x3: (Int, Int) => {
    val l: Int = x3._1; val r: Int = x3._2;
    __match.one(f(l, r))
}))
```

Nested pattern matches

```
p match {
  case Pair(Pair(ll, lr), r) => f(f(ll,lr), r)
}
```

are translated into nested calls to `flatMap`:

```
Pair.unapply(p).flatMap(x2: (Int, Int) => {
  val r: Int = x2._2;
  Pair.unapply(x2._1).flatMap(x4: (Int, Int) => {
    val ll: Int = x4._1; val lr: Int = x4._2;
    __match.one(f(f(ll, lr), r))
  })
})
```

Finally the pattern guards are translated into a call to the `guard` function that executes the by-name body of the case when the `cond` statement is satisfied.

5.2 DSL Intrinsification

DSL intrinsification maps, directly embedded versions of the DSL intrinsics, to their deep counterparts. The constructs that we translate (Figure 5.1) are: *i)* constants and free variables (§5.2.1), *ii)* DSL types (§5.2.2), and *iii)* DSL operations in the direct program (§5.2.3).

5.2.1 Constants and Free Variables

Constants. Constant values can be intrinsified in the deep embedding in multiple ways. They can be converted to a method call for each constant (e.g., $[\![\texttt{1}]\!] = \texttt{\_1}$), type (e.g., $[\![\texttt{1}]\!] = \texttt{liftInt(1)}$), or with a unified polymorphic function (e.g., $[\![\texttt{1}]\!] = \texttt{lift[Int](1)}$) that uses type classes to define behavior and the return type of `lift`.

In Yin-Yang we use the polymorphic function approach to translate constants

$$\frac{\Gamma \vdash c : T}{[\![c]\!] = \texttt{lift}[T](c)}$$

where c is a constant. We choose this approach as DSL frameworks commonly have a single IR node for all constants and it is easiest to implement such behavior with a single method.

The deep embedding can, as Scala supports type-classes, provide an implementation of `lift` that depends on the type of c. The DSL author achieves this by providing a type

class instance for lifting a set of types (defined by upper and lower bounds) of constants. This way different types can be reified in different ways.

In Yin-Yang we treat as constants:

1. *Scala literals* of all primitive types `Char`, `Short`, `Int`, `Long`, `Float`, and `Double`, as well as literals of type `String` (`"..."`), `Unit` (`()`), and `Null` (`null`).

2. *Scala object accesses* e.g., `Vector` in `Vector.fill` is viewed as a constant. This allows the DSL authors to re-define the default behavior of object access. Translating objects is optional as leaving their original semantics simplifies the implementation of the deep embedding, but requires special cases in type translation.

Free variables. Free variables are defined outside the EDSL scope, i.e, they are captured by a direct EDSL term. The DSL compiler can only access the type of free variables. The values of captured variables will become available only during evaluation (i.e., interpretation or execution after code generation). To inform the deep embedding about free variables they need to be treated specially by the translation. Further, the deep embedding needs to provide support for their interpretation.

Yin-Yang supports tracking free variables in the DSL scope by translating them into a call to the polymorphic function `hole[T]`. As the argument to `hole`, Yin-Yang passes a unique identifier assigned to each captured variable:

$$\frac{\Gamma \vdash x : T \quad x \texttt{ is a free variable} \quad id = \texttt{uid}(x)}{[\![x]\!] = \texttt{hole}[T](id)}$$

Method `hole` handles reification of free variables in the deep embedding. The identifier passed to `hole` is used determine the total order of the free variables. In case of Yin-Yang the total order is determined by sorting the unique identifiers in the ascending order.

The total order is used to relate the variables in the direct embedding and their representation in the deep embedding. In case of Yin-Yang, the sorted free variables are passed as arguments to the Scala function that is a result of DSL compilation. The deep embedding is aware of the order as each `hole` invocation has a unique variable identifier as the argument.

5.2.2 Type Translation

The *type translation* maps every DSL type in the, already virtualized, term body to an equivalent type in the deep embedding. In other words, the type translation is a

function on types. Note that this function is inherently DSL-specific, and hence needs to be configurable by the DSL author.

The type mapping depends on the input type and the context where the mapping is applied. For translations used in Yin-Yang, we need to distinguish between types in type-argument position, e.g. the type argument `Int` in the polymorphic function call `lam[Int, Int]`, and the others. To this end, we define a pair of functions $\tau_{\texttt{arg}}, \tau : \mathbb{T} \to \mathbb{T}$ where $\mathbb{T}$ is the set of all types and $\tau_{\texttt{arg}}$ and τ translate types in argument and non-argument positions, respectively. Having type translation as a function opens a number of possible deep embedding strategies. Alternative type translations can also dictate the interface of `lam` and `app` and other core EDSL constructs. Here we discuss the ones that we find useful in EDSL design:

The identity translation. If we choose τ to be the identity function and virtualization methods such as `lam`, `app` and `ifThenElse` to be implemented in the obvious way using the corresponding Scala intrinsics, the resulting translation will simply yield the original, direct EDSL program. Usages of this translation are shown in §6.2.

Generic polymorphic embedding. If instead we choose τ to map any type term T (in non-argument position) to `Rep[`T`]`, for some abstract, higher-kinded IR type `Rep` in the deep EDSL scope, we obtain a translation to a *finally-tagless, polymorphic* embedding [Carette et al., 2009, Hofer et al., 2008]. For this embedding, the translation functions are defined as:

$$\begin{aligned} \tau_{\texttt{arg}}(T) &= T \\ \tau(T) &= \texttt{Rep[}T\texttt{]} \end{aligned}$$

By choosing the virtualized methods to operate on the IR-types in the appropriate way, one obtains an embedding that *preserves well-typedness*, irrespective of the particular DSL it implements. We will not present the details of this translation here, but refer the interested reader to [Carette et al., 2009] and the description of the corresponding deep embedding for this translation (§6.3).

Eager inlining. In high-performance EDSLs it is often desired to eagerly inline all functions and to completely prevent dynamic dispatch in user code (e.g., storing functions into lists). This is achieved by translating function types of the form A`=>`B in the direct embedding into `Rep[`A`] => Rep[`B`]` in the deep embedding (where `Rep` again designates IR types). Instead of constructing an IR node for function application, such functions reify the whole body of the function starting with IR nodes passed as arguments. The effect of such reification is equivalent to inlining. This function representation is

used in LMS [Rompf and Odersky, 2012] by default and we use it in Figure 5.5. The translation functions are defined as:

$$\begin{aligned}
\tau_{\texttt{arg}}(T_1 \Rightarrow T_2) &= \texttt{error} \\
\tau_{\texttt{arg}}(T) &= T, \text{ otherwise} \\
\tau(T_1 \Rightarrow T_2) &= \texttt{Rep[}\tau_{\texttt{arg}}(T_1)\texttt{]} \Rightarrow \texttt{Rep[}\tau_{\texttt{arg}}(T_2)\texttt{]} \\
\tau(T) &= \texttt{Rep[}T\texttt{]}, \text{ otherwise}
\end{aligned}$$

This translation preserves well-typedness but rejects programs that contain function types in the type-argument position. In this case this is a desired behavior as it fosters high-performance code by avoiding dynamic dispatch. As an alternative to rejecting function types in the type-argument position the deep embedding can provide coercions from `Rep[`A`] => Rep[`B`]` to `Rep[`A`=>`B`]` and from `Rep[`A`=>`B`]` to `Rep[`A`] => Rep[`B`]`.

This translation disallows usage of curried functions in the direct programs. If we represent the curried functions as the polymorphic types then functions would appear in the type argument position. To re-introduce the curried functions we provide a translation with the modified rule for translation of functions:

$$\tau(T_1 \Rightarrow T_2) = \tau(T_1) \Rightarrow \tau(T_2)$$

which is correct only when the virtualization of functions and methods is disabled.

Custom types. All previous translations preserved types in the type parameter position. The reason is that the τ functions behaved like a higher-kinded type. If we would like to map some of the base types in a custom way, those types need to be changed in the position of type-arguments as well. This translation is used for EDSLs based on polymorphic embedding [Hofer et al., 2008] that use `this.T`, to represent the type `T`.

In Scala this translation can simply prepend each type with a prefix (e.g., `this`) so that the deep embedding can redefine it:

$$\begin{aligned}
\tau_{\texttt{arg}}(T) &= this.T \\
\tau(T) &= this.T
\end{aligned}$$

Unlike with the previous translations, where the type system of the direct embedding was ensuring that the term will type-check in the deep embedding, this translation gives

no guarantees. The responsibility for the correctness of the translation is on the DSL author. The deep embedding for this translation is presented in §6.5 and we applied this translation for embedding Slick [Typesafe] with great success (§11.2).

Untyped backend. If DSL authors want to avoid complicated types in the back-end (e.g., `Rep[T]`), the τ functions can simply transform all types to the `Dynamic` [Abadi et al., 1991] type. Giving away type safety can make transformations in the back-end easier for the DSL author.

Changing the translation. By simply changing the type translation, the EDSL author can modify behavior of an EDSL. For example, with the generic polymorphic embedding the EDSL will reify function IR nodes and thus allow for dynamic dispatch. In the same EDSL that uses the eager inlining translation, dynamic dispatch is restricted and all function calls are inlined.

5.2.3 Operation Translation

The *operation translation* maps directly embedded versions of the DSL operations into corresponding deep embeddings. To this end, we define a function `opMap` on terms that returns a deep operation for each directly embedded operation.

The `opMap` function in Scala intrinsifies direct EDSL body in the context of the deep embedding. `opMap` can be defined as a composition of two functions: *i)* function `inject` that inserts the direct EDSL body into a context where deep EDSL definitions are visible, and *ii)* `rebind` rebinds operations of the direct EDSL to the operations of the deep EDSL. Function `opMap` is equivalent to the composition of `inject` and `rebind` (written in Scala as `rebind andThen inject`).

Operation `rebind`. Operations in Scala are resolved through a *path*. For example, a call to the method `fill` on the object `scala.collection.immutable.Vector`

```
scala.collection.immutable.Vector.fill(1)(42)
```

has a path `scala.collection.immutable.List.fill`[3].

This phase translates paths of all operations, so they bind to operations in the deep embedding. The translation function can perform an identity operation, prefix addition, name mangling of function names, etc.

[3]Scala packages have a hidden prefix `_root_` that we omit for simplicity.

Operation `inject`. For Yin-Yang and polymorphic embeddings [Hofer et al., 2008], we chose to inject the body of our DSLs into the refinement of the Scala component that contains all deep DSL definitions:

$$[\![dsl]\!] = \textbf{new}\ \ dDSL\ \ \{\ \ \textbf{def}\ \ \texttt{main}() :\ \ \tau(T)\ \ =\ \ dsl\ \}$$

Term *dsl* is the direct DSL program after rebinding and *dDSL* is the component name that is provided by the DSL author that holds all definitions in the deep embedding, and τ is the type translation.

Alternatively the DSLs can be injected into the scope by importing all DSL operations from the object that contains the deep embedding definitions.

$$[\![dsl]\!] = \textbf{val}\ \ c\ \ =\ \ \textbf{new}\ \ dDSL;\quad \textbf{import}\ \ c.\_;\quad dsl$$

In both cases the corresponding `rebind` function can be left as an identity. If the deep embedding has all the required methods with corresponding signatures, all operations will rebind to the operations of the deep embedding by simply re-type-checking the program in the new context.

For example in a simple function application

```
Vector.fill(1)(42)
```

the injection will rebind the operation through the extension methods of Scala. The resulting code will contain wrapper classes that extend DSL types with deep operations:

```
VectorOps(lift(Vector)).fill(lift(1))(lift(42))
```

Finally, injection in the DSL scope allows the deep embedding operation to have additional implicit arguments. The implicit arguments the direct embedding are already resolved by type checking. This makes those arguments explicit after the translation, and leaves space for the deep embedding to introduce new implicit parameters. For example, the `fill` operation can be augmented with source information from the direct embedding and run-time type representation:

```
def fill[T](n: Rep[Int])(e: Rep[T])
  (implicit tt: TypeTag[T], sc: SourceContext): Rep[Vector[T]]
```

In Yin-Yang we chose the operation translations that closely match the structure of the direct embeddings. This allows the authors of the deep embedding to use the DSL itself for development of new DSL components. In this case the DSL author can use the deep interface augmented with implicit conversions that simplify lifting constants and calling operations. With such interface, the previous example resembles the direct embedding

```
Vector.fill(1)(42)
```

except for the abstraction leaks of the deep embedding (§4.2). This approach requires less code than "naked" AST manipulation [Visser, 2004], e.g:

```
VectorFill(Const(1), Const(42))
```

5.2.4 Translation as a Whole

To present the translation as a whole we use an example program for calculating $\sum_{i=0}^{n} i^{exp}$ using the vector EDSL defined in Figure 4.1. Figure 5.5 contains three versions of the program: Figure 5.5a depicts the direct embedding version, Figure 5.5b represents the program after type checking[4] (as the translation sees it), and Figure 5.5c shows the result of the translation.

In Figure 5.5c, we see two inconsistencies that were omitted from the translation rules for clarity:

- All translated methods are prepended with $. Character $ is added as this makes translation specific methods treated specially by the Scala tool-chain. Methods with $ are invisible in the deep embedding documentation, REPL, and IDE code auto-completion.
- The method `$hole` is not populated with type parameters but it has a call to `$tpe` added as a second argument. This is added as, in Scala one can not provide partial type arguments to the method and the idea of Yin-Yang, is to leave the definition of `$hole` open to the DSL authors. With `$tpe` added Scala's type-inference can infer the type parameters of method `$hole`. Similarly `$lift` is not provided by the type parameter, but the type is provided by the first argument.

In Figure 5.5c, on line 2 we see the DSL component in which the DSL program is injected. On line 3 we see how Scala's `if` is virtualized to `$ifThenElse`. On line 4, in the condition `n`, since it is a captured variable, is lifted to the call to `$hole(0,$tpe[Int])` as it is a captured variable and the identifier `0` as it shows as the first variable in the DSL program. On the same, line constant `0` is lifted to `$lift(0)`. On line 5 we see how the type of the `val` is translated to `Rep[Vector[Int]]` and how `val` is virtualized. On line 6 `vector.Vector` is lifted as a constant. On line 7 we see how variable `n` is replaced with a hole with the same identifier (`0`). This identifier communicates to the deep embedding that it is the same value. On line 8 the Scala function is virtualized to

[4]The Scala type-checker modifies the program by resolving identifiers to their full paths, adding implicit parameters, etc.

the `$lam` call. On line 9 the identifier `exp` is translated to `$hole(1,$tpe[Int])` (with the identifier `1`) as `n` appears as the second identifier in the program.

```
import vector._
import math.pow
val n = 100; val exp = 6;
vectorDSL {
  if (n > 0) {
    val v = Vector.range(0, n)
    v.map(x => pow(x, exp)).sum
  } else 0
}
```

(a) A program in direct embedding for calculating $\sum_{i=0}^{n} i^{exp}$.

```
1  val n = 100; val exp = 6;
2  vectorDSL {
3    if (n > 0) {
4      val v: Vector[Int] =
5        vector.Vector.range(0,n)
6      v.map[Int](x: Int =>
7        math.pow(x, exp)
8      ).sum[Int](
9        Numeric.IntIsIntegral)
10   } else 0
11 }
```

(b) The original program after desugaring and type inference.

```
1  val n = 100; val exp = 6;
2  new VectorDSL with IfOps with MathOps { def main() = {
3    $ifThenElse[Int](
4      $hole(0, $tpe[Int]) > $lift(0),{ // then
5      val v: Rep[Vector[Int]] = $valDef[Vector[Int]](
6        $lift(vector.Vector).range(
7          $lift(0), $hole($tpe[Int], 0)))
8      v.map[Int]($lam[Int, Int](x: Rep[Int] =>
9        $lift(math).pow(x, $hole($tpe[Int], 1))
10     ).sum[Int]($lift(Numeric).IntIsIntegral)
11   },{ // else
12     $lift(0)
13   })
14 }
```

(c) The Yin-Yang translation of the program from Figure 5.5b.

Figure 5.5 – Transformation of an EDSL program for calculating $\sum_{i=0}^{n} i^{exp}$.

5.2.5 Correctness

To completely conceal the deep embedding, all type errors must be captured in the direct embedding or by the translation, i.e., the translation must never produce an ill-typed program. Proving this property is verbose and partially covered by previous work. Therefore, for each version of the type translation we provide references to the previous work and give a high-level intuition:

- *The identity translation* ensures that well-typed programs remain well typed after

the translation to the deep embedding [Carette et al., 2009]. Here the deep embedding is the direct embedding with virtualized host language intrinsics.

- *Generic polymorphic embedding* preserves well-typedness [Carette et al., 2009]. Type `T` is uniformly translated to `Rep[T]` and thus every term will conform to its expected type.
- *Eager inlining* preserves well-typedness for programs that are not explicitly rejected by the translation. We discuss correctness of eager inlining in the appendix (§A.1) on a Hindley-Milner based calculus similar to the one of Carette et al. [Carette et al., 2009].

  For the intuition why type arguments can not contain function types, consider passing the increment function to the generic identity function:

  ```
  id[T => T](lam[T, T](x => x + 1))
  ```

  Here, the `id` function expects a `Rep` type but the argument is `Rep[T] => Rep[T]`.
- The *Dynamic* type supports all operations and, thus, static type errors will not occur. Here, the DSL author is responsible for providing a back-end where dynamic type errors will not occur.
- *Custom types* can cause custom type errors since EDSL authors can arbitrarily redefine types (e.g., `type Int = String`. Yin-Yang provides *no guarantees* for this type of the translation.

5.3 Translation in the Wider Context

We implemented Yin-Yang in Scala, however, the underlying principles are applicable in the wider context. Yin-Yang operates in the domain of statically typed languages based on the Hindley-Milner calculus, with a type system that is advanced enough to support deep DSL embedding. The type inference mechanism, purity, laziness, and sub-typing, do not affect the operation of Yin-Yang. Different aspects of Yin-Yang require different language features, which we discuss separately below.

Host-language requirements. The core translation is based on term and type transformations that require type introspection. To support Yin-Yang the host language must allow reflection, introspection, and transformation on types and terms. The translation can be achieved through a macro system or as a modification of the host-language. The translation can be achieved both at run-time and compile-time.

Semantic equivalence for prototyping and debugging. In order to support prototyping and debugging in the direct embedding, the direct EDSL program and the

corresponding deep EDSL program must always evaluate to the same value. This requirement allows the DSL end-user to prototype and debug programs in the direct embedding, and be certain that she will get the same output after translation.

Yin-Yang does not require that each sub-term of a DSL program evaluates to the same value. The reason for this "loose" definition of equivalence is that deep embeddings perform global program transformations to achieve better performance. Requiring semantic equivalence for each evaluation step of a program would preclude many optimizations (e.g., join reordering in relational algebra).

If there is a *semantic mismatch* [Czarnecki et al., 2004] between the two embeddings, e.g., the host language is lazy and the embedded language is strict, Yin-Yang can not be used for prototyping and debugging. In this scenario, the direct embedding can be implemented as stub that is used only for its user friendly interface and error reporting. Execution, and thus debugging, must be always performed in the deep embedding as the deep embedding provides adequate semantics.

6 Deep Embedding with Yin-Yang

The translation to the deep embedding assumes existence of an adequate deep embedding. This deep embedding should have an interface that corresponds to: *i)* methods emitted by language virtualization, *ii)* translation of constants and free variables, and *iii)* DSL operations defined in the direct embedding.

For each type translation, interface of the deep embedding and IR reification is achieved differently. In this section we describe how to define the deep embedding interface (§6.1) and achieve reification: *i)* for the identity translation (§6.2), *ii)* for the polymorphic embedding (§6.3), *iii)* for the polymorphic embedding with eager inlining (§6.4), and *iv)* for the embedding with custom types (§6.5). Finally, we discuss how the deep embedding is compiled and executed (§6.6). For conciseness, in all embeddings we omit the interface of pattern matching and virtualized universal methods.

6.1 Relaxed Interface of the Deep Embedding

The Yin-Yang translation requires a certain interface from the deep embedding. This interface, however, does not conform to the typical definition of an interface in object oriented programming languages. In most translation rules only the method name, number of type arguments, and number of explicit arguments is defined.

The number of implicit arguments and the return type are left open for the DSL author to define. The implicit parameters are added by the type-checker after the operation translation phase if they are present in the deep embedding. For example, if we define the `$ifThenElse` function as

```
def $ifThenElse[T](cond: Boolean, thn: => T, els: => T)(
  implicit tpe: TypeTag[T]): T = \\...
```

after translation, the Scala type-checker will provide the arguments that carry run-time

type information.

This feature can be used in the deep embeddings for different purposes. The ones that were used in DSLs based on Yin-Yang and LMS are: *i)* tracking type information, *ii)* tracking positions in the source code of the direct embedding, *iii)* changing the method return type based on its argument types, and *iv)* allowing multiple methods with the same erased signature (§4.2.7).

6.2 Embedding for the Identity Translation

The DSL embedding for the identity translation is not a deep embedding. The types are unmodified and, thus, can not represent the DSL intermediate representation. This translations is still interesting as a mechanism for instrumenting language constructs of Scala. The basic interface that contains all the language features, without any additional implicit parameters, is defined in Figure 6.1.

With this embedding the order of execution in all control flow constructs of Scala is preserved with by-name parameters. For example, in `$ifThenElse` both the `thn` and the `els` parameters are by-name and their execution order is defined by the implementation of the method. Similarly, `$whileDo`, `$doWhile`, `$try`, and `$lazyValDef` have their parameters by-name.

In Figure 6.1 method `hole` has a slightly different representation than what is presented in §5. Method `hole` has an additional parameter for the run-time type information (`tpe: TypeTag[T]`). This parameter is passed explicitly by Yin-Yang to allow the compiler to infer the type arguments of the method hole. This allows additional freedom for defining the interface of hole (see §6.5).

Using the identity translation for instrumentation. The DSL author can instrument Scala language features by re-defining virtualized methods. For example, collecting profiles of `if` conditions can be simply added by overriding the `$ifThenElse` method:

```
def $ifThenElse[T](cond: Boolean, thn: => T, els: => T)(
  implicit sc: SourceContext): T = {
  val (thnCnt: Long, elsCnt: Long) =
    globalProfiles.getOrElse(sc, (0, 0))

  globalProfiles.update(sc,
    if (cond) (thnCnt + 1, elsCnt)
    else (thnCnt, elsCnt + 1))
  if (cond) thn else els
}
```

Achieving the same effect with a facility like macros would require the DSL author to

know the reflection API of Scala which involves details about the Scala intermediate representation. Further, the DSL author would be required to write transformers of that intermediate representation.

After translation, features that are supported but not of interest can be removed with an interesting trick. If we implement each method in the deep embedding as a macro that inlines the method to its original language construct all abstraction overhead of virtualized methods disappears. For example, `$ifThenElse` can be returned to a regular if with

```
def $ifThenElse[T](cond: Boolean, thn: => T, els: => T): T =
  macro ifThenElseImpl[T]

def ifThenElseImpl[T](c: Context)(cond: c.Tree,
  thn: c.Tree, els: c.Tree): c.Tree = { import c.universe._
  q"if ($cond) $thn else $els"
}
```

Yin-Yang provides a Scala component that has methods of all language features overridden with macros that rewrite them to the original language construct. By using this component the DSL author can override functionality of individual language constructs without a need to redefine all other language features.

6.3 Polymorphic Embedding

With the generic polymorphic embedding we uniformly abstract over each type in the direct embedding. This abstraction can be used to give different semantics (thus polymorphic) to the deep programs. In the context of the deep DSLs the most common semantics is reification of the intermediate representation.

In Figure 6.2 we show the basic interface for the generic polymorphic embedding. In trait `PolymorphicBase` the type `R[+T]`[1] is the abstraction over the types in the direct embedding. This type is covariant in type `T` and therefore the abstract types have the same subtyping relation as the types in the direct embedding.

In this embedding both function definition and function application are abstracted over. Method `$lam` converts functions from the host language into the functions in the embedded language and method `$app` returns functions from the embedded language into the host language. This way the DSL author has complete control over the function definition and application in the deep embedding.

[1] Some deep embeddings call this type `Rep` or `Repr`. We use the name `R` for its conciseness. In languages with local type inference this type is omnipresent in method signatures and makes them longer.

```
trait FunctionsInterface {
  // only a subset of function arities is displayed
  def $app[U](f: () => U): () => U
  def $lam[U](f: () => U): () => U

  def $app[T_1, U](f: T_1 => U): T_1 => U
  def $lam[T_1, U](f: T_1 => U): T_1 => U
}

trait IdentityInterface with FunctionsInterface {
  // constants and captured variables
  def $lift[T](v: T): T
  def $hole[T](id: Long, tpe: TypeTag[T]): T
  def $tpe[T]: TypeTag[T]

  // control structures
  def $ifThenElse[T](cond: Boolean, thn: => T, els: => T): T
  def $return(expr: Any): Nothing
  def $whileDo(cond: Boolean, body: => Unit): Unit
  def $doWhile(body: => Unit, cond: Boolean): Unit
  def $try[T](body: => T, catches: Throwable => T, fin: => T): T
  def $throw(t: Throwable): Nothing

  // variables
  def $valDef[T](init: T): T
  def $lazyValDef[T](init: => T): T
  def $varDef[T](init: T): T
  def $read[T](init: T): T
  def $assign[T](lhs: T, rhs: T): Unit
}
```

Figure 6.1 – Interface of the identity embedding.

```
trait PolymorphicBase { type R[+T] }

trait GenericFunctionsBase extends PolymorphicBase {
  // only a subset of function arities is displayed
  def $app[U](f: R[() => U]): () => R[U]
  def $lam[U](f: () => R[U]): R[() => U]

  def $app[T1, U](f: R[T1 => U]): R[T1] => R[U]
  def $lam[T1, U](f: R[T1] => R[U]): R[T1 => U]
}

trait PolymorphicInterface extends GenericFunctionsBase {
  // constants and captured variables
  def $lift[T](v: T): R[T]
  def $hole[T](id: Long, tpe: TypeTag[T]): R[T]
  def $tpe[T]: TypeTag[T]

  // control structures
  def $ifThenElse[T](
    cnd: R[Boolean], thn: => R[T], els: => R[T]): R[T]
  def $return(expr: R[Any]): R[Nothing]
  def $whileDo(cnd: R[Boolean], body: => R[Unit]): R[Unit]
  def $doWhile(body: => R[Unit], cond: R[Boolean]): R[Unit]
  def $try[T](
    body: => R[T], catches: R[Throwable => T], fin: => R[T]): R[T]
  def $throw(e: R[Throwable]): R[Nothing]

  // variables
  def $valDef[T](init: R[T]): R[T]
  def $lazyValDef[T](init: => R[T]): R[T]
  def $varDef[T](init: R[T]): R[T]
  def $read[T](init: R[T]): R[T]
  def $assign[T](lhs: R[T], rhs: R[T]): R[Unit]
}
```

Figure 6.2 – Interface of the generic polymorphic embedding.

Compared to standard deep embeddings in Scala (e.g., LMS) the DSL author should provide the interface compatible with Yin-Yang. With Yin-Yang the DSL operation are added to `R[T]` types in the same manner. In addition the DSL author should: *i)* provide lifting for Scala objects and *ii)* transform the bodies of deep embedding operations.

Objects in the deep embedding. Objects in the deep embedding should correspond to the translated objects in the direct embedding. Since we treat objects as constants they are handled by the `$lift` method.

An object from the direct embedding should be represented with an appropriate `lift` method and a set of extension methods that represent its operations. For example, to support the `println` method on the `Predef` object the deep embedding must introduce a lifting construct and an extension method on `R[Predef.type]`:

```
def $lift(c: Predef.type): R[Predef.type] = Const(Predef)
implicit class PredefOps(predef: R[Predef.type]) {
  def println(p: R[Any]): R[Unit] = \\...
}
```

To preserve the original API for objects in the deep embedding one can introduce a shortcut for the `Predef` object:

```
val Predef = lift[Predef.type](scala.Predef)
```

This way the deep embedding can be used in a similar way to the direct embedding for the purpose of developing internal DSL components.

Operations in the direct embedding. Implementation of the deep embedding operations depends on which rules of virtualiztion are applied. If we apply method virtualization in the DSL programs the bodies of the deep embedding operations should be transformed with the same transformation. For example, an identity method in the deep embedding

```
def id[T](v: R[T]): R[T] = Identity(v)
```

should be transformed into

```
def id[T]: R[T => T] = $lam[T]((x: R[T]) => Identity(v))
```

In practice, the deep DSL operations only reify the deep program so tracking function application of these methods introduces superfluous IR nodes. Further, this transformation convolutes implementation of the deep embedding. For these reasons, in most of

```
trait InliningFunctionsBase extends PolymorhpicBase {
  def $app[U](f: () => R[U]): () => R[U]
  def $app[T_1, U](f: R[T_1] => R[U]): R[T_1] => R[U]

  def $lam[U](f: () => R[U]): () => R[U]
  def $lam[T_1, U](f: R[T_1] => R[U]): R[T_1] => R[U]
}
```

Figure 6.3 – Interface of the polymorphic embedding with eager inlining.

the DSLs it is common to disable method virtualization. This way the domain-specific operations are always executed in the host language.

6.4 Polymorphic Embedding with Eager Inlining

Polymorphic embedding with eager inlining differs from the generic polymorphic embedding in the way host language functions are translated. With this type of embedding functions are left unmodified in the host language and therefore executed during DSL compilation. Effectively, this way of treating functions always inlines all functions in the deep embedding.

With the type translation for eager inlining, there are two possibilities for the DSL author: *i)* use the full language virtualization but disallow curried functions, and *ii)* to completely disable virtualization of functions and allow curried functions. In both of these cases the DSL author must implement the interface `PolymorphicInterface` from Figure 6.2, however, the interface for functions is different.

Eager inlining with function virtualization. If the full virtualization is used the deep embedding should provide an interface shown in Figure 6.3. This way the user can track function applications and definitions in the deep embedding with a drawback that curred functions are not allowed.

Eager inlining without function virtualization. Without virtualization the interface for functions in the deep embedding is not necessary. The deep embedding will use the functions from the host language. This kind of embedding is the primary choice of DSLs based on LMS[2].

[2]LMS supports both generic polymorphic embedding and embedding with eager inlining. However, in LMS DSLs [Rompf and Odersky, 2012] and tutorials eager inlining is more common.

```
trait CustomFunctionsBase {
  type Function0[U]
  type Function1[T, U]
  def $app[U](f: Function0[U]): () => U
  def $app[T_1, U](f: Function1[T_1, U]): T_1 => U

  def $lam[U](f: () => U): Function0[U]
  def $lam[T_1, U](f: T_1 => U): Function1[T_1, U]
}
```

Figure 6.4 – Interface of the embedding with custom types. The DSL author can arbitrarily override each type in the embedding.

6.5 Embedding With Custom Types

In the embedding with custom types the DSL author is required to redefine every type of the direct embedding in the deep embedding. The overriding is achieved with the abstract types of Scala. For example, the type `scala.Boolean` can be overridden with a type `scala.Int` (as it is in the C language) inside of the DSL component with:

```
type Boolean = scala.Int
```

Interface of these embeddings is equivalent to the interface of the identity embedding (Figure 6.1) except that all types need to be abstract. Further, Scala functions can not be directly converted into abstract types so all functions in the deep embedding must be represented with their nominal equivalents (e.g., `Int => Int` must be represented as `Function1[Int, Int]`).

An example of the interface for functions is presented in Figure 6.4 where type definitions (`type Function0[U]` and `type Function1[T,U]`) make the function types abstract.

A common pitfall with naming abstract types is using only their type name (e.g., `Function0`). The problem arises when two different types (with different paths) have the same name. In these cases the DSL author must use the full type path as the abstract type. Yin-Yang can be configured to add a prefix to the types with their full path or only to their name. In all examples, for conciseness reasons, we use the translation that does not use the full path but only the type name.

In the embedding with custom types a common way to reify the DSL IR is to define abstract types with the IR nodes of the deep embedding. In this scheme, in order to preserve well-typedness after the translation each IR node must satisfy the interface of the type from the direct embedding. The methods in the overridden type all methods can perform reification and still return the correct program.

```
trait DSLBase {
  trait Exp // base class for all nodes
}
trait BooleanDSL extends DSLBase {
  type Boolean = BooleanOps
  trait BooleanOps with Exp {
      def &&(y: Boolean): Boolean = BooleanAnd(this, y)
      def ||(y: Boolean): Boolean = BooleanOr(this, y)
      def unary_!: Boolean = BooleanNot(this)
    }
  }

  case class BooleanAnd(lhs: Boolean, rhs: Boolean)
    extends BooleanOps
  case class BooleanOr(lhs: Boolean, rhs: Boolean)
    extends BooleanOps
  case class BooleanNot(lhs: Boolean)
    extends BooleanOps
}
```

Figure 6.5 – Overriding semantics of `Boolean` with the reification semantics for the deep embedding.

In Figure 6.5 we show how one reifies operations on the `Boolean` type. All nodes in the IR usually have a common supertype (`Exp` in the example). Then the abstract type for `Boolean` is overridden with a trait (`BooleanOps`) that redefines its operations. These redefined operations can now perform reification of the program in the deep embedding. With this embedding all IR nodes of type `Boolean` extend `BooleanOps` and therefore all method implementations have valid types.

The types in the deep embedding can not be expressed as a type function and therefore defining the interface of `$hole` and `$lift` is different than the other embeddings. Here, for each type in the direct embedding there needs to be a definition that maps the type to the deep embedding.

One way to achieve such type map is to use function overloading in Scala. The deep embedding should have one `$hole` and `$lift` method for each type in the direct embedding. For example, lifting the type `Int` is achieved with:

```
  def $hole(id: Long, tpe: TypeTag[scala.Int]): this.Int =
    new Hole(sym, v) with IntOps
  def $lift(v: scala.Int): this.Int =
    new Const(v) with IntOps
```

DSL author is free to redefine types arbitrarily thus giving the deep embedding different

```
trait Executable {
  def execute(args: Any*): Any
}
```

Figure 6.6 – The trait for that Yin-Yang uses to execute the deep embedding.

semantics. With different semantics the DLS embedding can perform pretty printing of code or provide a wrapper for another DSL. For further information on such embeddings see [Hofer et al., 2008] and §11.2.

6.6 The Yin-Yang Interface

Yin-Yang does not require an exact interface for the translation to the deep embedding and therefore there are no `traits` that a DSL author must extend in order to satisfy the translation. The interface is defined by method names and their argument counts, emitted by virtualization and by the types dictated by the type translation.

After the translation, a program in the deep embedding should be executed and should produce the same result[3] as the direct embedding. Execution of a program should happen transparently to the DSL, i.e., it should completely resemble the execution in the direct embedding.

Execution of the deeply embedded program depends on values of all variables that were captured by the direct embedding. Yin-Yang, after the core translation, must assure that the deep embedding has access to all the values that were replaced by holes during translation. The DSL compiler or interpreter must also be able to determine to which hole a passed value belongs.

To achieve this, Yin-Yang requires that the deep embedding implements the `Executable` interface (Figure 6.6). This interface contains a single method `execute` that is used for execution of the DSL at host language run-time. Yin-Yang simply calls the method `execute` on the translated DSL passing it the free variables captured by the direct embedding.

Yin-Yang invokes `execute` by passing all free variables sorted in ascending order by the unique identifier that was assigned to each free variable. Since the identifiers passed to `hole` are strictly increasing, the DSL author can uniquely map arguments to the holes they belong to. For example, the DSL program in the example translation (Figure 5.5c) has two captured variables (`n` and `exp`) with identifiers `0` and `1` respectively. Yin-Yang will, after translation, invoke this DSL with

[3]The result can be different in case of floating-point operations executed on different back-end platforms, however this difference should not change decisions that are made based on the result.

```
dsl.execute(n, exp).asInstanceOf[Int]
```

This way of invoking a DSL is convenient for the DSL author as one needs to implement a single method for evaluating the DSL, however, it is not optimal. Passing arguments as variable arguments requires boxing and storing of arguments in a sequence (`Seq`). This introduces an additional level of indirection and imposes run-time overheads.

To avoid this overhead, the DSL authors can define additional execute methods in the DSL body with a more specific types in the method. During type checking, Scala's overload resolution always chooses the most specific method and thus avoids boxing and the intermediate sequence. For example, if a DSL from the example translation ((Figure 5.5c)) would implement a method with a signature

```
def execute(v0:Int, v1:Int): Int
```

this method would be invoked instead of the method with variable length arguments.

The deeply embedded DSLs can generate and compile code in the host language, other languages, or be interpreted. We have abstracted over these types of DSLs with a simple interface. With this approach, all compilation decisions are left to the DSL author. This simplifies the Yin-Yang framework, however, it complicates management of DSL compilation in presence of multiple DSLs.

In the case where multiple DSL frameworks simultaneously invoke compilation of a program, it is possible that they exhaust memory resources of the running process. For example, each compilation in Scala requires loading the whole compiler and requires significant amounts of memory. When multiple frameworks use compilation without coordination, the system resources are easily exhausted.

Yin-Yang currently does not provide a way to coordinate compilation between different DSLs. Resolving this problem has been studied before in context of JIT compilation [Arnold et al., 2000, Kulkarni, 2011] and it falls out of the scope of this work.

6.7 Defining a Translation for a Direct DSL

The direct-to-deep embedding translation is defined as a macro that accepts a single by-name parameter that represents a body of a DSL program. For example, the `vectorDSL` from Figure 5.5 is defined as

```
def vectorDSL[T](body: => T): T = macro vectorDSLImpl[T]
```

where `vectorDSLImpl` represents the macro implementation.

```
object YYTransformer {
  def apply[C <: Context, T](c: C)(
    dslType: c.Type,
    tpeTransformer: TypeTransformer[c.type],
    config: YYConfig): YYTransformer[c.type, T]
}
```

Figure 6.7 – Interface to the Yin-Yang translation.

The macro implementation calls into a single transformer (`YYTransformer`) that implements the whole translation. The `YYTransformer` is configured: *i)* with the type of a DSL component that implements the deep embedding (e.g., `la.VectorDSL`) , *ii)* with a type transformer (e.g., generic polymorphic transformer), and *iii)* with a configuration object (i.e., defines whether to virtualize functions, methods, lift Scala objects, etc.). Interface of the object that creates a `YYTransformer` is shown in Figure 6.7.

With the `YYTransformer` object the method `vectorDSLImpl` is implemented as:

```
def vectorDSLImpl[T](c: Context)(body: c.Expr[T]): c.Expr[T] =
  YYTransformer[c.type, T](c)(
    typeOf[la.VectorDSL],
    new EagerInliningTT[c.type](c),
    YYConfig(virtualizeFunctions = false)(body)
```

here the type of the `VectorDSL` is provided in with `typeOf` construct, `EagerInliningTT` represents the translation for polymorphic embedding with eager inlining, and `YYConfig` defines that functions should not be virtualized.

7 DSL Reification at Host-Language Compile Time

To integrate the deep embedding tightly with the host-language, it is necessary to have the DSL IR reified at host language compilation-time. With the compile-time reified IR, the DSL can perform analysis and report errors during host language compilation (§8.2). Further, DSLs that do not use run-time values for optimization can be completely compiled at host-language compile time. Their generated code is then integrated with the IR of the host language with no overhead (§9).

Reifying the DSL IR after the translation requires executing generated code after the direct-to-deep translation. Since the translation happens during program type-checking, the executable code for the translated DSL does not exist—only the host-language IR is available. To execute this code during type-checking there are two possibilities:

- **Use compilation.** Since all the captured variables are replaced by holes (§5.2.1), the deep embedding can be compiled down to executable code separately from the rest of the host-language program. Then the deep embedding binary can be executed to obtain the domain-specific IR.

- **Use interpretation.** Use the interpreter of the host language trees to obtain the domain-specific IR. Since the deep DSL has no captured variables, it is possible to perform interpretation.

Compilation has an upfront cost but then, the execution is optimal. On the other hand interpretation has no up-front cost but the execution is slower due to interpretation overheads. For Scala, and for different sizes of DSL programs, we evaluate performance of compilation and interpretation for DSL compilation at host-language compile time (§7.3).

7.1 Reification by Compilation

The DSL program that is being compiled to the executable during host language type-checkingj can always be compiled because:

- All captured variables have been converted into calls to `$hole` by the translation.
- Class and object definitions that would require complicated interpretation are forbidden by the virtualization phase.
- Usage of definitions (i.e., classes, traits, objects, and methods) that are compiled in the same compilation unit is impossible. The direct embedding definition does not contain equivalents for these objects and they are rejected by *language restriction*. Language restriction is described in §8.1.

Given that the translated DSL body is independent of the current compilation unit, it is possible to compile it separately and produce the executable file. For the DSL body, the compilation pipeline is executed to the last phase starting from type-checking. Then the DSL body can be executed to acquire an instance of the DSL with its IR reified.

In Yin-Yang, for the purpose of compile-time reification, Yin-Yang uses the `eval` construct in Scala. The `eval` function accepts the Scala IR and compiles it to bytecode, loads that bytecode, and executes it to acquire the DSL IR.

In case of the JVM, the reification with compilation has a relatively large overhead. Compiling the Scala IR to bytecode and loading that bytecode is costly. Then, the bytecode that is loaded must be executed in the JVM interpreter as it is executed for the first time. Finally, the code is executed only once in order to reify the DSL program and discarded afterwards so JIT compilation never happens.

7.2 Reification by Interpretation

An alternative to compilation is interpretation of the host-language trees that represent the deep embedding. With interpretation, an *interpreter* interprets the IR emitted by the direct-to-deep translation to obtain an instance to a reified DSL at host-language compile time. Since the DSL trees do not depend on the definitions in the current compilation unit, it is possible to interpret them.

Since the Yin-Yang translation greatly simplifies the executed trees, we use a specialized interpreter. In this interpreter we treat only a subset of Scala that is emitted by the DSL translation. Further, the interpreter invokes all functions in the program with reflection. This way the execution of the DSL operations is achieved by executing pre-compiled code and the interpreter executes the DSL programs.

With the specialized interpreter proper care must be taken with higher-order functions. The DSL operations that accept functions as arguments must accept function instances while the function bodies must be executed in the interpreter. To this end, we pass function instances that call back into the interpreter code to pre-compiled operations.

7.3 Performance of Compile-Time Reification

We compare compilation and interpretation on synthetic DSL programs based on LMS that vary in size from 0 to 500 reified IR nodes. In the benchmark we use a mix of IR nodes that includes control flow nodes, arithmetic nodes, and constant nodes. For each program we measure the time it takes to instantiate the DSL compiler and reify the IR for a program. Benchmarks use the methodology, as well as hardware and software platform defined in §A.2.

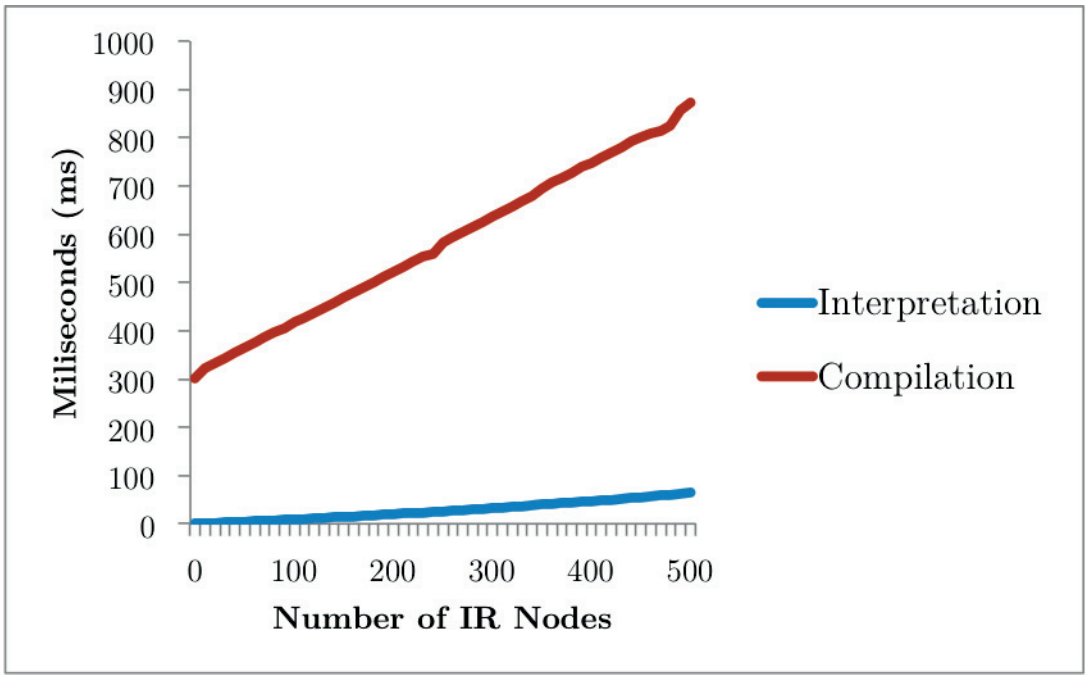


Figure 7.1 – Time to reify the IR of DSL programs by means of compilation and interpretation.

In Figure 7.1 we see that reification by compilation-and-execution is outperformed by interpretation in all cases. Compilation introduces a constant overhead of 0.3 seconds that comes from instantiating a new Scala compiler that compiles the Scala ASTs to bytecode. After the initial overhead the reification speed is 175 IR nodes per second. The slow rate of reification is due to compilation overheads and the execution on the JVM platform. The freshly loaded bytecode is executed only once, and thus it will run in the JVM interpreter. On the other hand, Scala AST interpretation has a small initial overhead of 0.3 ms and reifies on average 1577 IR nodes per second.

To this end we chose interpretation as the default choice for compile-time reification of DSLs in Yin-Yang.

8 Improving Error Reporting of Embedded DSLs

Error reporting of the deep embedding does not blend into the host language in two ways: *i)* the DSL is not able to restrict generic language features of the host language and errors can happen at run-time (§4.2.4) and *ii)* domain-specific errors can only be reported at host-language run-time (§4.2.5). This chapter shows how once "we broke the ice" with using the host-language reflection, improving error reporting becomes a simple addition and how the DSL author can be completely agnostic of the host language reflection API. In §8.1 we show how restricting host language constructs is achieved by omitting operations in the deep embedding and in §8.2 we explain error reporting at host language compile-time.

8.1 Restricting Host-Language Constructs

The direct DSL programs can contain well-typed expressions that are not supported by the deep embedding. Often, these expressions lead to unexpected program behavior (§4) and we must rule them out by reporting meaningful and precise error messages to the user.

We could rule out unsupported programs by relying on properties of the core translation. If a direct program contains unsupported expressions, after translation those expressions can not be bound to deep embedding constructs, and the programs become ill-typed. We could reject unsupported programs by, simply, reporting type checking errors. Since, the direct program is well-typed and the translation preserves well-typedness (§5.2.5) all type errors after translation must be due to unsupported operations.

Unfortunately, naively restricting the language by detecting type-checking failures is leaking internal information about the deep embedding. The reported error messages will contain virtualized language constructs and types. This is not desirable since users should not be exposed to the internals of the deep embedding.

Yin-Yang avoids leakage of the deep embedding internals in error messages by performing an additional verification step that, in a fine grained way, checks if a method from the direct program exists in the deep embedding. This step traverses the tree generated by the core translation and verifies for each method call, if it correctly type-checks in the deep embedding. If the type checking fails, Yin-Yang reports two kinds of error messages:

- Generic messages for unsupported methods:

```
List.fill(1000, Vector.fill(1000,1)).reduce(_+_)
^
Method List.fill[T] is unsupported in VectorDSL.
```

- Custom messages for unsupported host language constructs:

```
try Vector.fill(1000, 1) / 0
^
Construct try/catch is unsupported in VectorDSL.
```

With Yin-Yang, the DSL author can arbitrarily restrict virtualized constructs in an embedded language by simply omitting corresponding method definitions from the deep embedding. Due to the additional verification step, all error messages are clearly shown to the user. This allows easy construction of embedded DSLs that support only a subset of the host language, without the need to know the Scala reflection API.

8.2 Domain-Specific Error Reporting at Compile Time

The domain-specific error reporting is performed on the reified DSL program. In order to achieve error reporting at host-language compile time, Yin-Yang must first reify the DSL IR. To this end, Yin-Yang uses the techniques described in §7 to reify the program.

Once the program is reified, Yin-Yang calls the `staticallyCheck` method from Figure 8.1 on the DSL component. This method accepts as an argument the `Reporter` interface that the DSL author can use to report messages on the host language console. The DSL author can use the `Reporter` to report errors (the `error` method), warnings (the `warning` method), and to provide informational messages (the `info` method).

All methods in the `Reporter` interface accept the `pos: Option[Position]` argument for displaying where the information should be displayed. If the position argument is not passed, the error will be displayed without a position. If it is provided, the message will be displayed for the source code at the position, in the same manner as the host language would.

The interface that is provided for error reporting is much simpler than the Scala reflection

```
trait Position {
  def source: File
  def line: Int
  def column: Int
}

trait Reporter {
  def info(pos: Option[Position], msg: String): Unit
  def warning(pos: Option[Position], msg: String): Unit
  def error(pos: Option[Position], msg: String): Unit
}

trait StaticallyChecked {
  def staticallyCheck(c: Reporter): Unit
}
```

Figure 8.1 – Interface for domain-specific error reporting in the deep embedding.

equivalent. The API is simplified for error reporting in domain-specific languages and does not require the DSL author to comprehend the Scala internals. For the values of the `Position` interface the DSL author can choose an adequate implementation (e.g., one presented by Rompf et al. [Rompf et al., 2013a]).

9 Reducing Run-Time Overhead in the Deep Embeddings

In this chapter we first discuss run-time overhead introduced by the deep embedding (§9.1). Then, we measure run-time overhead of existing deep embeddings and compare them (§9.2). Finally, we show how DSL reification at compile-time can completely remove run-time overhead (§9.3).

9.1 Introduction: Runtime Overheads

The deep embedding necessarily introduces overhead. As the DSL compilation is performed at host language run-time it incurs additional execution steps. We categorize overhead related to run-time DSL compilation in the following categories:

- **DSL program reification** is the time necessary to reify the program.
- **DSL compilation** is the process after reification that applies domain-specific optimizations and generates a final version of the code.

Since the deep programs are compiled at run-time, captured variables are seen by the DSL as compile-time constants. Those captured variables are, then, further used in compilation decisions. The DSL compiler, without user modifying the program, can not distinguish between captured variables and constants. Let's examine a simple Slick DSL program that queries the database for finding all cars with price higher than `minPrice`:

```
def expensiveCars(minPrice: Double): Seq[Car] =
  TableQuery[Car].filter(c => c.price > minPrice).result
```

Here the query must be compiled with every call to `expensiveCars` method, as for each call to the method, the generated SQL will be different. In Slick, both reification and query compilation will happen at every execution of the method `expensiveCars`.

Without translation or modification of the DSL programs, the deep embedding must, at every program execution, at least reify the program. Without users modifying the code, DSL compilers can not distinguish between constants and captured variables. Therefore, in order to see if recompilation is required, the deep embedding must reify the IR every time and verify that all DSL compilation-time constants are the same as in the previous runs.

Some DSLs solve the problem of reification on every execution, by making explicit parametrization of captured variables. This way the captured variables are marked by the programmer and the DSL program is written outside of the scope to avoid re-reification of the program. For example, Slick supports query templates where the user first defines a query that is compiled once, and then uses it later. For example, definition of the `expensiveCars` query is:

```
val expensiveCarsCompiled = Compiled((minPrice: Rep[Double]) =>
  TableQuery[Car].filter(c => c.price > minPrice))

def expensiveCars(minPrice: Double): Seq[Car] =
  expensiveCarsCompiled(minPrice).result
```

and its later usage would not recompile the query on every execution. However, we see that the user must augment the program in order to avoid re-compilation.

In all deep embeddings we have seen, both reification and compilation are performed on every execution. Compilation on every execution, however, could be avoided by comparing the reified IR in the current executions to the previously stored IRs. Then, compilation can be avoided by re-using the already compiled code from a cache. This way, the deep DSLs could avoid full recompilation at every execution.

Since re-compilation can be avoided in §9.2 we measure only the run-time overhead that comes from reification. Then, in §9 and §9.3.2, we describe a translation-based solution for minimizing run-time overhead in the deep embedding.

9.2 Measuring Run-Time Overheads

We measured the cost of reification in three different IRs: *i)* the core IR of LMS, *ii)* the IR of the Slick DSL, and the *iii)* a synthetic IR that only builds a simple object graph and has no reification overhead for maintaining symbol tables etc. The benchmark first initializes the compiler for Slick and LMS, and then, it runs a mix of synthetic lines of code which contain control flow statements and simple expressions. Benchmarks use the methodology, as well as the hardware and software platform, defined in §A.2.

In the benchmark the initialization phase of reification, instantiates the compiler in case

Table 9.1 – The initialization cost and cost of reification per reified IR node for the simple IR, Slick, and LMS.

EDSL	Initialization (μs)	Reification (μs/IR node)
Simple IR	0	0.05
Slick	2	1
LMS	1	4.9

of LMS and the `Table` object in case of Slick. This phase is relatively cheap compared to reification. We notice that the simple IR is far less costly than the IR in the existing DSLs. The reason for this difference is that DSLs, besides simple reification, do other operations during reification (e.g., tracking source information, tracking types, and simplifications of the IR). Finally, we see that reification in real-world DSLs costs between 1 μs and 4.9 μs per IR node.

For long running computations, the reification costs are negligible, however, the overhead is relevant when DSLs are used in tight loops. To put these numbers in a perspective, for the time of reifing one line of code (approximately 5 IR nodes) the same hardware platform can copy a 100000 element array of integers, sort a 400 element array of integers, or transfer 3 KB over the 1 GB/s network.

9.3 Reducing Run-Time Overheads

Before we describe our solution, we will categorize DSLs based on their interaction with the free variables:

- **One stage DSLs.** This category of DSLs that never uses run-time values for compilation decisions. This category of DSLs could be compiled during host-language compilation. For this category of DSLs we enable compilation at host-language compile time (§9.3.1).
- **Two-stage DSLs.** This category makes compilation decisions (e.g., optimizations) based on the values of free variables. These DSLs benefit from compilation at host-language run time. For this category of DSLs we provide a translation that removes the reification overhead (§9.3.2).

Even in multi-stage DSLs, run-time compilation is not always necessary. If the DSL program does not capture interesting values, or the captured values are not of interest for compilation decisions, compilation can be executed at host language compile-time. For this case we automate the decision whether a DSL program is treated as one-stage or two-stage. We introduce a workflow for deciding a compilation-stage for each compiled DSL program (§9.3.3).

```
trait Generator

trait CodeGenerator extends Generator {
  def generate(holes: String*): String
}

trait TreeGenerator extends Generator {
  def generate(c: Context)(holes: c.Tree*): c.Tree
}
```

Figure 9.1 – Interface for generating code at host-language compile time.

9.3.1 Avoiding Run-Time Overheads for One Stage DSLs

For this category of DSLs we exploit compile-time reification of Yin-Yang to generate code at host-language compile time. After the compile-time reification of the program Yin-Yang demands from the DSL to generate code. The generated code is then inlined, into the block where the original direct DSL program was and that direct DSL program is discarded.

To achieve compile-time compilation the DSL author must extend one of the two traits from Figure 9.1. The `CodeGenerator` requires from the DSL author to return a string with the generated code. The code generator must be hygienic [Kohlbecker et al., 1986], i.e., generated code must not accidentally capture variables from the surrounding scope. Also, all free variables must be replaced with adequate values from the `holes` parameter. Similarly `TreeGenerator` generates directly Scala trees. Both trees and strings are supported, since some DSL authors might prefer working with quasi-quotes and trees, while the others prefer strings.

9.3.2 Reducing Run-Time Overheads in Two-Stage DSLs

For DSLs that are compiled in two stages, ideally, we would reify and compile the DSL only in the first execution. Consecutive executions would reuse the result of the first DSL compilation for all future executions. Also, the DSL logic would decide on program recompilation based on the values that are passed to its `execute` method.

Ideally, we would store the result of DSL instantiation and reification into a global value definition that is computed only once. Instead of invoking execute on the original direct program, Yin-Yang would invoke `execute` on that global definition. The invocations of the `execute` method would take care of recompilation when input values change. For example, a simple direct program

```
vectorDSL { Vector.fill(1000, 1) }
```

would be converted into a definition

```
val prog$<UID> = new VectorDSL {
  def main() = lift(Vector).fill(lift(1000), lift(1))
}
```

and the original DSL would be replaced with

```
prog$<UID>.execute[Vector[Int]]()
```

In case of Yin-Yang and Scala creating a global value definition is not possible: Scala macros can introduce only local definitions. Therefore the unique value `prog$<UID>` can not be defined from the translation macro.

To overcome this issue Yin-Yang gives DSL programs a unique identifier and stores them in a global, concurrently accessible, data structure. With this modification the previously defined program is translated into

```
val prog = YYStorage.programs.computeIfAbsent(<UID>, { _ =>
  new VectorDSL {
    def main() = lift(Vector).fill(lift(1000), lift(1))
})
prog.execute[Vector[Int]]()
```

where `computIfAbsent` assures thread-safe access and instantiates the DSL only once: the DSL is instantiated when the unique identifier is not found in a map which happens only on the first execution. All issues related to concurrency that might arise only on recompilation inside the `execute` method are left for the DSL author to resolve.

9.3.3 Per-Program Decision on the DSL Compilation Stage

Compilation of two-stage DSLs still introduces a large compilation overhead in the first run and minor overhead for fetching the DSL in consecutive runs. This is not always necessary as two-stage DSLs can be compiled at host-language compile-time when their compilation decisions do not depend on captured values. This happens in one of the two cases: *i)* no values are captured by the DSL program (even though they could have been), or *ii)* captured values can not be used for optimization.

In order to compile these two categories of DSLs at host-language compile time, Yin-Yang must get information from the DSL compiler whether captured values are of interest for compilation. To this end Yin-Yang provides an interface `Staged` from Figure 9.2. The method `compileTimeCompiled` is invoked during host-language compilation to determine the stage of a program. The trait `Staged` depends on the trait `Generator` as, in order to perform DSL compilation the DSL must be able to generate code.

```
trait Staged { self: Generator =>
  def compileTimeCompiled: Boolean
}
```

Figure 9.2 – Interface for determining the compilation stage of a DSL at host-language compilation time. To allow greater flexibility in the deep embedding the concrete program is not passed as a parameter to the function `compileTimeCompiled`. It is left for the DSL author to declare how the program is fetched in subcomponents of `Staged`.

If the method `compileTimeCompiled` returns `true` the DSL, will be compiled at compile time with no run-time overhead (as described in §9.3.1). If the method returns `false`, the DSL will be compiled at run-time (as described in §9.3.2). The decision about compilation stage is domain-specific: it is made in the DSL component by analyzing usage of holes in the reified body.

10 Putting It All Together

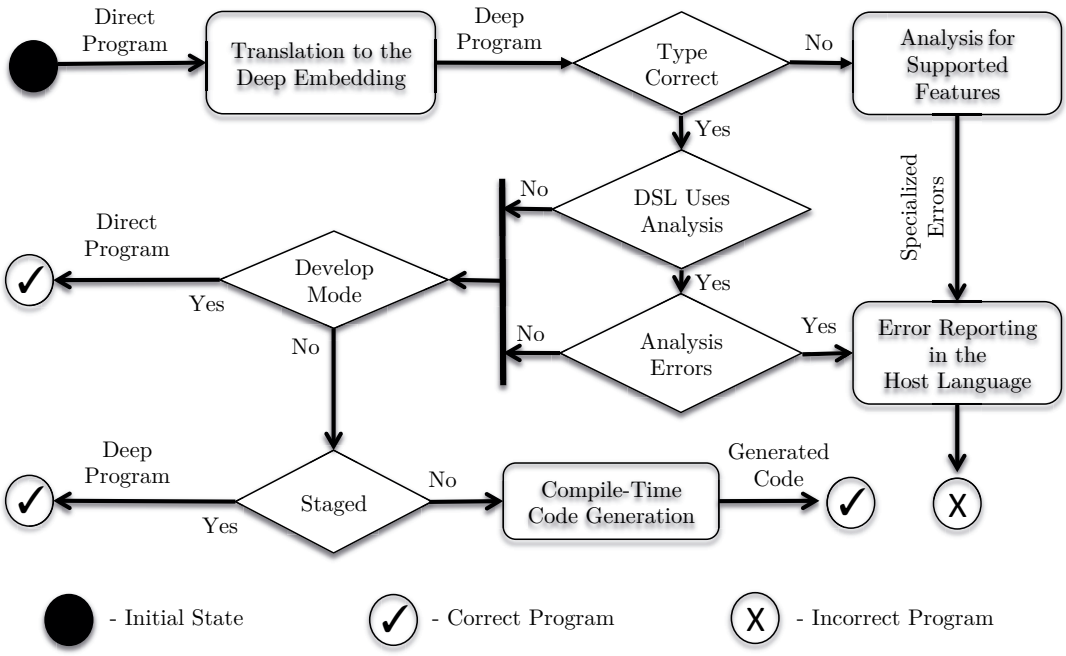


Figure 10.1 – Overview of the Yin-Yang framework: the workflow diagram depicts what happens to the DSL program throughout the compilation pipeline.

In this chapter we provide an overview over the individual steps in Yin-Yang operation. Yin-Yang makes decisions based on the development mode, the user written programs, and the DSL definition. These decisions use reflection to achieve debugging, language restriction, domain-specific error reporting, and DSL compilation at host-language compile time.

Figure 10.1 shows the workflow around the core translation that allows all benefits of the deep embedding during program development and hides all of its abstraction leaks.

The direct program is first translated into the deep embedding and type checked. At this point, since the translation is guaranteed to preserve well typedness §5.2.5, the type errors can arise from incorrect implementation of the deep embedding, or from usage of construct that do not exist in the deep embedding. Assuming that the implementation is correct the only possible source of type errors are unsupported language constructs.

At this point, if the program is incorrect the specialized analysis for supported features is performed. The unsupported features in the direct embedding are detected and reported with the precise position and specialized error messages. The reported errors use the standard reporting mechanism of the host language that works in all environments (e.g., IDEs, REPLs, and on standard error output). After supported feature analysis, the compilation is terminated.

In the correct program, the next decision checks whether the DSL uses domain-specfic analysis. This check is performed by looking into the hierarchy of the DSL component for the `StaticallyChecked` trait. If the DSL uses domain-specific analysis, the DSL is reified at host-language compile time and the `staticallyCheck` method is called. The errors are reported at host-language compile time with the standard host-language reporting mechanism. In case of errors the host-language compilation is terminated.

If there are no domain-specific errors or the DSL does not use domain-specific analysis, Yin-Yang checks whether the DSL is used in development mode. If the development mode is active, the original direct program is returned and the compilation succeeds. Returning the original program allows the DSL end-user to perform debugging in the direct embedding by inspecting the host language values.

In the non-development mode, the DSL is checked for the stage it should be compiled in. DSLs that do not extend `Generator` can not output host-language code and must be staged. They are directly sent for run-time compilation.

For DSLs that expose code generation there are two possibilities:

- The program does not capture variables so compilation should not be staged.
- The program captures variables. At this point, the decision on the compilation stage is based on the DSL and the program. The method `compileTimeCompiled` is invoked on the reified DSL to determine the compilation stage.

If the DSL is not staged, the code generation is invoked on the reified DSL (method `generate`). The code is then inlined in the DSL invocation site and the original direct embedding is discarded. The compile-time compilation completely removes the run-time overhead of the deep embedding. In case the DSL is staged, the deep program is augmented with the logic for program caching to avoid reification on every program execution.

11 Evaluation and Case Studies

This chapter evaluates the improvements of Yin-Yang based DSLs compared to standard deep embedding. The interface improvement is evaluated by counting deep embedding related annotations in the test suites of OptiML and OptiGraph EDSLs that are obviated by Yin-Yang (§11.1). Then, this chapter demonstrates the ease of adopting Yin-Yang for the existing deep EDSL Slick [Typesafe] (§11.2).

11.1 No Annotations in the Direct Embedding

To evaluate the number of obviated annotations related to the deep embedding we implemented a direct embedding for the OptiGraph EDSL (an EDSL for graph processing), and used the direct EDSL for OptiML. We implemented the all example applications of these EDSLs with the direct embedding. The 21 applications we implemented have 1284 lines of code.

To see the effects of using the direct embedding as the front-end, we counted the number of deep embedding related annotations that were used in the example applications. The counted annotations are `Rep` for types and `lift(t)` for lifting literals when implicit conversions fail. In 21 applications the direct embedding obviated 96 `Rep[T]` annotations and 5 `lift(t)` annotations. Overall, 12% of lines of code contained annotations related to the deep embedding.

11.2 Case Study: Yin-Yang for Slick

Slick is a deeply embedded Scala EDSL for database querying and access. Slick is not based on LMS, but still uses `Rep` types to achieve reification. Slick has a sophisticated interface for handling reification of tuples. This interface is, however, complicated as it adds additional type parameters and implicit values to main operations. An example of the Slick method is presented in the following snippet:

```
trait Query[T] {
  def map[S](projection: T => S): Query[S]
  def length: Int

  def sortBy[S](projection: T => S)
    (implicit ord: Ordering[S]): Query[T]
  def sorted(implicit ord: Ordering[T]): Query[T]

  def groupBy[S](f: T => S): Query[(S, Query[T])]
  def union(q2: Query[T]): Query[T]
  def leftJoin[S](q2: Query[S]): JoinQuery[T, S]
}
```

Figure 11.1 – Excerpt from the direct interface for Slick.

```
  def map[F, G, T](f: E => F)
   (implicit shape: Shape[_ <: FlatShapeLevel, F, T, G])
   : Query[G, T, C]
```

To improve the complicated interface of Slick, we used Yin-Yang. Since the deep embedding of Slick already exists, we first designed the new interface (direct embedding). The new interface has dummy method implementations, since semantics of different database back-ends is hard to be mapped to Scala. Thus, this interface is used only for user friendly error reporting and documentation. The interface is completely new, covers all the functionality of Slick, and consists of only 70 lines of code. The excerpt of the interface is presented in Figure 11.1.

Slick's complicated method signatures do not correspond to the simple new interface. In order to preserve backward compatibility, the redesign of Slick to fit Yin-Yang's core translation was not possible. We addressed this incompatibility by adding a simple wrapper for the original deep embedding of Slick that matches the signature required by Yin-Yang. The wrapper contains only 240 lines of straightforward code. We show how the wrapper methods link the new interface with the original deep interface in the following two methods:

```
  def map[S](projection: YYRep[T] => YYRep[S]): YYQuery[S] =
    YYQuery.fromQuery(query.map(underlyingProjection(projection))(
    YYShape.ident[S]))

  def filter(projection: YYRep[T] => YYRep[Boolean]): YYQuery[T] =
    YYQuery.fromQuery(
      query.filter(underlyingProjection(projection))(
      BooleanRepCanBeQueryCondition))
```

In the `map` method we see the type `YYQuery`. This type is the type that we translate to from the direct embedding (we use the translation with custom types). The `fromQuery`, method converts the original deep query into the intermediate type `YYQuery`. The `map` operation is performed on the deep version of the query and the `YYShape.ident[S]` is the example of the method that creates implicit arguments that satisfy the type checker in the original interface. In method `filter` we see `BooleanRepCanBeQueryCondition` which is another instance of the implicit argument that satisfies type checking of the original interface. These implicit parameters are the culprit for the complicated interface of Slick.

Building a new interface of Slick required only 120 hours of development. The new front-end passes 54 tests that cover the most important functionality of Slick. In the new interface, all error messages are idiomatic to Scala and resemble typical error messages from the standard library.

Performance improvements. As Yin-Yang minimizes the overhead related to runtime compilation we evaluated the performance of the new interface. The performance improvements are allowed by the core translation that converts all captured variables, into calls to `$hole`. With this information the deep embedding backend can convert user written queries into pre-compiled database queries. In the deep embedding, the user has to modify the queries to achieve the same functionality.

We measured the compilation overhead introduced by Slick. We used synthetic benchmarks that represent common short queries for selection, insertion, and updates. For selection with one predicate Slick introduces the compilation overhead of 380 μs per execution, for simple insertion 10 μs per execution, and for update with a single predicate an overhead of 370 μs per execution. With Yin-Yang, the overhead is smaller than 1 μs in all cases.

12 Related Work

Yin-Yang is a framework for developing embedded DSLs in the spirit of Hudak [Hudak, 1996, 1998]: embedded DSLs are *Scala libraries* and DSL programs are just *Scala programs* that do not, in general, require pre- or post-processing using external tools. Yin-Yang translates directly embedded DSL programs into finally-tagless deep embeddings [Carette et al., 2009]. Our approach supports (but is not limited to) polymorphic [Hofer et al., 2008] deep embeddings, and – as should be apparent from the examples used in this thesis – is particularly well-adapted for deep EDSLs using an LMS-type IR [Rompf et al., 2013a,b].

We will classify related work to the approaches that improve DSL author experience in external DSLs, approaches that use the shallow embedding as an interface for the deep embedding (§12.2) and approaches that try to improve the deep embedding interface (§12.3).

12.1 Improving DSL-Author Experience in External DSLs

The most noticeable approach to improving DSL-author experience in external DSLs are language workbenches [Fowler, 2005]. Workbenches such as Spoofax [Kats and Visser, 2010] and Rascal [Klint et al., 2009, van der Storm, 2011] provide a DSL-author-friendly declarative language, intended for the DSL authors to specify their languages in a high-level way. The language workbenches then generate large parts of the language ecosystem based on the provided specification. Language workbenches allow to automatically generate the parser, type-checker, name binding logic [Konat et al., 2013], IDE support [Kats and Visser, 2010], and debuggers (for a detailed overview of language workbenches see work by Erdweg et al. [Erdweg et al., 2013]).

12.2 Shallow Embedding as an Interface

Sujeeth et al. propose Forge [Sujeeth et al., 2013a], a Scala based meta-EDSL for generating equivalent shallow and deep embeddings from a single specification. DSLs generated by Forge provide a common abstract interface for both shallow and deep embeddings through the use of abstract `Rep` types. A shallow embedding is obtained by defining `Rep` as the identity function on types, i.e. `Rep[T] = T`.

A DSL end-user can switch between the shallow and deep embeddings by changing a single flag in the project build. Unfortunately, the interface of the shallow embedding generated by Forge remains cluttered with `Rep` type annotations. Finally, some plain types that are admissible in a shallow program, may lack counterparts among the IR types of the deep embedding. This means that some seemingly well-typed DSL programs become ill-typed once the Forge's transition from the shallow to the deep embedding is made. This forces users to manually fix type errors in the deeply embedded program after transition.

In Forge, the following categories of can occur after the transition:

- Using a method from the shallow embedding that is not present in the deep embedding. The types in the shallow embedding correspond to types in Scala and therefore can support more methods than the deep embedding. After transition to the deep embedding, these methods would not be detected.
- Scala's weak type conformance does not hold in the deep embedding. If weak type conformance is applied in the shallow embedding, after translation, users will get a cryptic type error.
- Accidentally using functions that are not in the shallow embedding. For example, if a user passes a higher-order function that does not use `Rep` types it will work in the shallow embedding but fail in the deep embedding:

```
list.map((x: Int) => x + 1) // fails in the deep embedding
```

Project Lancet [Rompf et al., 2013c] by Rompf et al. and work of Scherr and Chiba [Scherr and Chiba, 2014] interpret Java bytecode to extract domain-specific knowledge from directly embedded DSL programs compiled to bytecode. These solutions are similar to Yin-Yang in that the, direct embedding is translated to the deep embedding, however, there are several differences:

- Bytecode interpretation happens after the host language has finished compilation to produce bytecode. Therefore, it is hard to incorporate error reporting in the host-language from this stage.

- The proposed solutions do not support compile-time code generation. They always postpone compilation to runtime.
- These approaches do not support language restriction. This feature could be executed, but its implementation is not straight forward. For example, logical operators are represented as conditionals in bytecode and distinguishing between a conditional and a logical operation is not possible.

Awesome Prelude [Lokhorst, 2012] proposes replacing all primitive types in Haskell with type classes that can then be implemented to either construct IR or execute programs directly. This allows to easily switch between the two embeddings while the type classes ensure equivalent type checking. Unfortunately, this approach does not extend easily to native language constructs, and requires changing the type signatures of common functions. In the following example we show the modified signatures of the `Eq` type class:

```
class Eq j a where
  (==) :: (BoolC j) => j a -> j a -> j Bool
  (/=) :: (BoolC j) => j a -> j a -> j Bool
```

In the Haskell's `Eq` the constraint `BoolC` and variable `j` do not exist.

Kansas Lava [Gill et al., 2011] is a Haskell EDSL for hardware specification. In Kansas Lava Gill et al. propose a design flow where the hardware is first specified in a Haskell standard library and then it is manually translated into the shallow embedding that uses the applicative functor `Signal` to guarantee that the circuit can be synthesized. Finally, the DSL end-user translates the program to the deep embedding to generate hardware. In this workflow the translation is not automated and the DSL end-user must do it manually for each program. Here, the interface of the shallow embedding is not equivalent to the direct embedding as the difference in the interface is used to verify if the code can be synthesized.

12.3 Improving the Deep Embedding Interface

Scala Virtualized [Rompf et al., 2013a] allows usage of host language constructs on DSL types. This is achieved by translating all language constructs to method calls and letting the DSL author override them with alternative semantics. The same idea was proposed by Carette et al. and Yin-Yang builds upon it. In Yin-Yang, we also translate the types with a configurable type translation and integrate the translation into the host language.

Heeren et. al [Heeren et al., 2003, Hage and Heeren, 2007] introduce extensible error reporting to type systems. With their approach, the library author can provide custom error messages for specific usage patterns of their libraries and DSLs. This can be used in the deep embedding to improve convoluted and incomprehensible error messages. With

respect to improved error messages in Yin-Yang, the approach of Heeren et. al is more advanced as it can improve error reporting beyond the problems that arise from the deep embedding.

Svenningsson and Axelson [Svenningsson and Axelsson, 2013] propose combining a small deeply embedded core with a shallow front-end. The shallow front-end uses the small core to provide all language features to DSL end-users: it is a layer over the core deep embedding that is used to improve its interface. In their approach, however, the shallow interface is still modified to hide the IR construction. These modifications are visible to the user through the interface of DSLs and require additional effort from the user to comprehend the deep embedding artifacts.

Part II
Automating Deep Embedding Development

13 Translation of Direct EDSLs to Deep EDSLs

In the first part we used translation from programs written in the direct embedding into the deep embedding. This arguably simplifies life for EDSL users by allowing them to work with the interface of the direct embedding. However, the EDSL author still needs to maintain synchronized implementations of the two embeddings. This is a tedious and error prone task that can be automated: all information necessary for building the deep embedding front-end is already present in the direct embedding.

To alleviate this issue, we introduce a way to automatically generate the deep embedding from the implementation of the direct embedding. We use the interface and the implementation of the direct embedding to generate the deep embedding. The direct interface is used to generate the corresponding deep interface. The direct bodies are translated and used to generate the deep bodies with equivalent semantics.

In this chapter we first discuss the subset of Scala that we support in the direct embedding (§13.1), then we define the parts of the deep embedding generation that are customizable by the DSL author (§13.2). We show a case study of deep embedding generation for LMS based DSLs (§13.3). Finally, we discuss the interactions between deep-embedding generation and domain-specific optimizations in §13.4.

13.1 Supported Language Constructs

The deep-embedding generation supports a subset of Scala in the direct embedding. In this section we define what are the language features that are allowed in the direct interface: *i)* as type definitions, *ii)* as member definitions, and *iii)* as expressions.

Type definitions. Definitions of new types are limited in the direct embedding. The inclusive list of type definitions that are supported are:

- Traits without initialization logic with only abstract and final members.
- Top-level classes that can inherit only previously defined traits. Nested classes, or class inheritance is not allowed.
- Objects that can inherit admissible traits. Object nesting is allowed only in other objects.

Member definitions. Direct-to-deep translation supports a subset of member definitions. It allows all kinds of value binders (i.e., values, lazy values, and variables). Then, it allows methods that follow the rules imposed by type definitions. Type members are not allowed.

Expressions. In method bodies the direct embedding translation supports all of the constructs listed in (§5). In addition the translation supports type aliases and import statements as these constructs get resolved during type checking: they do not affect the translation.

In the context of deep embedding generation, the program translation needs to be slightly modified. The direct embedding method definitions are written inside type definitions and should support the `this` expression. To provide a DSL author defined behavior of `this`, we add an additional virtualization rule:

$$[\![\texttt{this}]\!] = \$\texttt{self}$$

13.2 Customizable Code Templates

In the deep embedding parts of the implementation are strictly fixed while other parts can vary. The interface of the deep embedding, in a weaker sense of the word (§6), is strictly dictated by the translation and can not be modified. The member definitions and the organization of Scala components, however, can be custom for each DSL.

Generating the interface. Since the interface is prescribed by translation, the layout of methods, their arguments, types of arguments can not be modified. We provide code generation for generic polymorphic embeddings as shown in (§6.3), and polymorphic embeddings with eager inlining (§6.4). The DSL author can not alter this part of code generation.

Template inputs. For the method implementation generation it is necessary to provide necessary information to the DSL author so she can define: *i)* an IR node naming scheme,

ii) a policy about additional implicit parameters in the deep embedding, *iii)* how to lower an IR node to its low-level implementation, and *iv)* how to handle custom annotations in the direct embedding.

To allow flexibility for generating member definitions we provide access to the *definition signature*: this includes all arguments, their modifiers, their types, and all annotations. Further we provide information about the chain of owner definitions. This information gives all the flexibility to the DSL author to define their embedding scheme.

We also provide the DSL author with the code of the translated bodies of the direct embedding members. This information can be used for defining lowering transformations that transform high-level IR nodes into their low-level representation.

Support for lowering is beneficial:

- As it minimizes the required development effort in the deep embedding. Implementation of all DSL operations is already defined in the direct embedding and DSL authors need not repeat it.

- As it guarantees correctness of the deep embedding given that: *i)* the program translation is correctly implemented and *ii)* that the core language in the direct embedding and the deep embedding have the same semantics.

Member definitions. With the input information DSL authors can define their deep embeddings. The DSL authors are allowed to add implicit parameters (e.g., for type information, source information, etc.) and method bodies. In the method bodies DSL authors can make transformations based on the translated code of the direct embedding, or make decisions based on annotations.

Components and internals organization. Deep embeddings we use, are organized in Scala components with different possibilities for organization and internal interfaces. Although, the DSL authors could require custom internal interfaces, we do not allow this.

The DSL internals should be organized so that: *i)* that the DSL end-user never sees the internals, *ii)* that the DSL author has a possibility to transform code by using the deep embedding interface, *iii)* and there are no hidden problems with the design (e.g., long compilation times). Based on best practices in the LMS and Delite frameworks we designed a scheme of components that fulfills all of the requirements for Yin-Yang based DSLs. If the DSL authors want to modify the scheme, they are required to change the framework.

13.3 A Case Study: Generating LMS Based DSLs

In this section we demonstrate an example of how the deep embedding is generated. We will generate a deep embedding that follows the common patterns for DSL design with LMS. Besides LMS, the same technique has been successfully used for generating a deep embedding for the current version of the query compiler LegoBase [Klonatos et al., 2014].

Generation happens in two steps: *i)* we generate high-level IR nodes and methods that construct them through a systematic conversion of methods declared in a direct embedding to their corresponding methods in the deep embedding (§13.3.1), and *ii)* we exploit the fact that method implementations in the direct embedding are also direct DSL programs. Reusing our core translation from §5, we translate them to their deep counterparts (§13.3.3).

13.3.1 Constructing High-Level IR Nodes

Based on the signature of each method, we generate the *case class* that represents the IR node. Then, for each method we generate a corresponding method that instantiates the high-level IR nodes. Whenever a method is invoked, in the deep EDSL, instead of being evaluated, a high-level IR node is created.

Figure 13.1 illustrates the way of defining IR nodes for `Vector` EDSL. The case classes in the `VectorOps` trait define the IR nodes for each method in the direct embedding. The fields of these case classes are the callee object of the corresponding method (e.g., `v` in `VectorMap`), and the arguments of that method (e.g., `f` in `VectorMap`).

13.3.2 Special Processing Based on Annotations

Here we show how the method bodies can be modified based on annotations from the deep embedding. Our motivating example are the effect annotations. In practice we also used the same principle for partial evaluation and for generating local optimizations.

Deep embedding should, in certain cases, be aware of side-effects. The EDSL author must annotate methods that cause side-effects with an appropriate annotation. To minimize the number of needed annotations, we use Scala FX [Rytz et al., 2012]. Scala FX is a compiler plugin that adds an effect system on top of the Scala type system. With Scala FX the regular Scala type inference also infers the effects of expressions. As a result, if the direct EDSL is using libraries which are already annotated, like the Scala collection library, then the EDSL author does not have to annotate the direct EDSL. Otherwise, there is a need for manual annotation of the direct embedding by the EDSL author. Finally, the Scala FX annotations are mapped to the corresponding effect construct in LMS.

```
trait VectorOps extends SeqOps with
  NumericOps with Base {
  // elided implicit enrichment methods. E.g.:
  //   Vector.fill(v, n) = vector_fill(v, n)

  // High level IR node definitions
  case class VectorMap[T:Numeric,S:Numeric]
    (v: Rep[Vector[T]], f: Rep[T] => Rep[S])
    extends Rep[Vector[S]]
  case class VectorFill[T:Numeric]
    (v: Rep[T], size: Rep[Int])
    extends Rep[Vector[T]]

  def vector_map[T:Numeric,S:Numeric]
    (v: Rep[Vector[T]], f: Rep[T] => Rep[S]) =
      VectorMap(v, f)
  def vector_fill[T:Numeric]
    (v: Rep[T], size: Rep[Int]) =
    VectorFill(v, size)
}
```

Figure 13.1 – High-level IR nodes for `Vector` from Figure 4.1.

Figure 13.2 shows how we automatically transform the I/O effect of a `print` method to the appropriate construct in LMS. As the Scala FX plugin knows the effect of `System.out.println`, the effect for the `print` method is inferred together with its result type (`Unit`). Based on the fact that the `print` method has an I/O effect, we wrap the high-level IR node creation method with a call to `reflect`, which is an effect construct in LMS to specify an I/O effect [Rompf et al., 2011]. In effect, all optimizations in the EDSL will have to preserve the order of `println` and other I/O effects. We omit details about the LMS effect system; for more details cf. [Rompf et al., 2011].

13.3.3 Lowering High-Level IR Nodes to Their Low-Level Implementation

Here we show how the translated body can be used for creating transformers. We demonstrate the concept on the lowering transformation that converts high-level nodes to their low-level implementations.

Having domain-specific optimizations on the high-level representation is not enough for generating high-performance code. In order to improve the performance, we must transform these high-level nodes into their corresponding low-level implementations. Hence, we must represent the low-level implementation of each method in the deep EDSL. After creating the high-level IR nodes and applying domain-specific optimizations, we

```
class Vector[T: Numeric](val data: Seq[T]) {
  // effect annotations not necessary
  def print() = System.out.print(data)
}
trait VectorOps extends SeqOps with
  NumericOps with Base {
  case class VectorPrint[T:Numeric]
    (v: Rep[Vector[T]]) extends Rep[Vector[T]]
  def vector_print[T:Numeric](v: Rep[Vector[T]]) =
    reflect(VectorPrint(v))
}
```

Figure 13.2 – Direct and deep embedding for Vector with side-effects.

transform these IR nodes into their corresponding low-level implementation. This can be achieved by using a *lowering* phase [Rompf et al., 2013b].

Figure 13.3 illustrates how the invocation of each method results in creating an IR node together with a lowering specification for transforming it into its low-level implementation. For example, whenever the method `fill` is invoked, a `VectorFill` IR node is created like before. However, this high-level IR node needs to be transformed to its low-level IR nodes in the lowering phase. This delayed transformation is specified using an `atPhase(lowering)` block [Rompf et al., 2013b]. Furthermore, the low-level implementation uses constructs requiring deep embedding of other interfaces. In particular, an implementation of the `fill` method requires the `Seq.fill` method that is provided by the `SeqLowLevel` trait.

```
trait VectorLowLevel extends VectorOps
  with SeqLowLevel {
  // Low level implementations
  override def vector_fill[T:Numeric]
    (v: Rep[T], s: Rep[Int]) =
    VectorFill(v, s) atPhase(lowering) {
      Vector.fromSeq(Seq.fill[T](s)(v))
    }
}
```

Figure 13.3 – Lowering to the low-level implementation for Vector generated from the direct embedding.

Generating the low-level implementation is achieved by transforming the implementation of each direct embedding method. This is done in two steps. First, the expression given as the implementation of a method is converted to a Scala AST of the deep embedding by core translation of Yin-Yang. Second, the code represented by the Scala AST must be injected back to the corresponding trait. To this effect, we implemented

Sprinter [Nikolaev], a library that generates correct and human readable code out of Scala ASTs. The generated source code is used to represent the lowering specification of every IR node.

13.4 DSL Development with Code Re-Generation

The automatic generation of deep embeddings reduces the amount of boilerplate code that has to be written and maintained by EDSL authors, allowing them to instead focus on tasks that can not be easily automated, such as the implementation of domain-specific optimizations in the deep embedding. However, automatic code generation is not a silver bullet. Hand-written optimizations acting on the IR typically depend on the structure of the later, introducing hidden dependencies between such optimizations and the direct embedding. Care must be taken in order to avoid breaking optimizations when changing the direct embedding of the EDSL.

While these optimizations are not re-generated; only the components that correspond to the interface and the IR nodes are modified. Therefore, the DSL author is only responsible for maintaining analysis and optimizations in the deep embedding. A change in the direct embedding interface affects only optimizations related to that change.

The DSL is composed out of Scala components that the DSL author defines. Besides the components that are the output of the deep embedding generator, the DSL author can add arbitrary components to the DSL, e.g., for optimizations. In the following example we see a component that rewrites all `VectorFill(Const(0), s)` nodes with `VectorZero(s)` nodes:

```
trait VectorZeroTransformer extends ForwardTransformer {
  val IR: VectorDSL with LMSCore
  import IR._
  override def transformStm(stm: Stm): Exp[Any] = stm match {
    case TP(_, VectorFill(Const(0), s)) => VectorZero(s)
    case _ => super.transformStm(stm)
  }
}
```

We use this example to show what happens when the direct embedding is changed and deep embedding re-generated. In the given transformer, only the names and the arguments of IR nodes are related to the direct embedding. In this particular case, the DSL author defined node names as capitalized names of its owner definitions concatenated together. In this scheme a change of the name in any definition in the owner chain (e.g., `Vector` or `Map`) or change of the number of arguments would be break the `VectorZero` transformer. Given code, however is independent of the other constructs in the direct embedding and their change would not affect this transformer.

One way to avoid the dependency on definition names is to give custom node names to methods through annotations. For the `Vector.fill` method this could be achieved as following:

```
@IR("Fill") def fill[T](v: T, s: Int)
```

With this scheme the change of the name of `Vector` and `fill` would not affect optimizations with a drawback that the direct embedding code is modified.

13.5 Evaluation

To evaluate automatic deep EDSL generation for OptiML, OptiQL, and Vector, we compared Yin-Yang to Forge [Sujeeth et al., 2013a], a Scala based meta-EDSL for generating both direct and deep EDSLs from a single specification. Forge already contained specifications for OptiML and OptiQL.

To avoid re-implementing the direct embedding of OptiML and OptiQL programs we modified Forge to generate the direct embedding from its specification. Then, we used the modification to generate the direct embeddings from the existing Forge based EDSL specifications. On the generated direct embedding we applied our automatic deep generation tool to produce its deep counterparts. For all three EDSLs, we verified that tests running in the direct embeddings behave the same as the tests for the deep embeddings.

In Table 13.1, we give a line count comparison for the code in the direct embedding, Forge specification, and deep embedding for three EDSLs: *i) OptiML* is a Delite-based EDSL for machine learning, *ii) OptiQL* is a Delite-based EDSL for running in-memory queries, and *iii) Vector* is the EDSL shown as an example throughout this thesis. We are careful with measuring lines-of-code (LOC) with Forge and the deep EDSLs: we only count the parts which are generated out of the given direct EDSL.

Overall, Yin-Yang requires roughly the same number of LOC as Forge to specify the DSL. This can be viewed as positive result since Forge relies on a specific meta-language for defining the two embeddings. Yin-Yang, however, uses Scala itself for this purpose and is thus much easier to use. In case of OptiML, Forge slightly outperforms Yin-Yang. This is because Forge supports meta-programming at the level of classes while Scala does not.

We did not compare the efforts required to specify the DSL with Yin-Yang and Forge. The reason is twofold:

- It is hard to estimate the effort required to design a DSL. If the same person designs a single DSL twice, the second implementation will always be easier and take less time. On the other hand, when multiple people implement a DSL their skill levels

Table 13.1 – LOC for direct EDSL, Forge specification, and deep EDSL.

EDSL	Direct	Forge	Deep
OptiML	1128	1090	5876
OptiQL	73	74	526
Vector	70	71	369

can greatly differ.

- Writing the direct embedding in Scala is arguably simpler than writing a Forge specification. Forge is a Delite-specific language and uses a custom preprocessor to define method bodies in Scala. Thus, learning a new language and combining it with Scala snippets must be harder than just writing idiomatic Scala.

13.6 Related Work

Forge [Sujeeth et al., 2013a] uses a new embedded language with string pre-processing to generate the deep embedding. Compared to our approach, Forge additionally provides meta-programming facilities to generate methods for their front-end that is based on `Rep` types. In Forge, however, the declarative language is a deeply embedded DSL instead of native Scala, the method bodies are represented as strings, and method bodies are not guaranteed to be type correct in both deep and shallow embeddings.

Language workbenches [Erdweg et al., 2013] use a declarative language to define a new language, generate its ASTs, IDE integration, parser etc. Here we use the direct embedding augmented with user annotations to define a language representation that already supports IDE integrations, has a parser, and the rest of compilation pipeline. Our approach can be viewed as a language workbench for embedded domain-specific languages.

14 Dynamic Compilation of DSLs

Domain-specific language (DSL) compilers often require knowledge of values that are known only at program run-time to perform optimizations. For example, in matrix-chain multiplication, knowing matrix sizes allows choosing the optimal multiplication order [Cormen et al., 2001, Ch. 15.2] and in relational algebra knowing relation sizes is necessary for choosing the fastest order of join operators [Selinger et al., 1979]. Consider the example of matrix-chain multiplication:

```
val (m1, m2, m3) = ... // matrices of unknown size
m1 * m2 * m3
```

In this program, without knowing the matrix sizes, the DSL compiler can not determine the optimal order of multiplications. There are two possible orders `(m1*m2)*m3` with an estimated cost `c1` and `m1*(m2*m3)` with an estimated cost `c2` where:

```
c1 = m1.rows*m1.columns*m2.columns+m1.rows*m2.columns*m3.rows
c2 = m2.rows*m2.columns*m3.columns+m1.rows*m2.rows*m3.columns
```

Ideally we would change the multiplication order at run-time only when the condition `c1 > c2` changes and not the individual inputs. Matrix-chain multiplication requires global transformations based on conditions outside the program scope. For this task *dynamic compilation* [Auslander et al., 1996] seems as a good fit.

Yet, dynamic compilation systems—such as DyC [Grant et al., 2000]—have shortcomings. They are controlled only with user annotations, and that do not give the user complete control over specialization. Further they are not well suited for development of the deep embeddings. In deep embeddings:

- The optimizations based on run-time-captured values happen out of the scope of user programs throughout the DSL compiler. To enable dynamic compilation with existing systems all data structures and functions in the DSL compiler would have

to be marked as run-time constants.

- The user should not know about that run-time compilation is happening. Yet, with these approaches, she would have to annotate values, as well as DSL constructs, for specialization purposes.

We propose a dynamic compilation system aimed for domain-specific languages where:

- DSL authors declaratively, at the definition site, state the only the values that are of interest for dynamic compilation (e.g., array and matrix sizes, vector and matrix sparsity). The values of interest are marked with a parametric type (`SD[T]`) while the rest of the DSL compiler stays unmodified. The DSL compiler is designed such that all operations available on a native Scala type (`T`) are also available on the type `SD[T]`, thus, the DSL authors can freely use the terms of type `SD`.

  The type system propagates the `SD` typed terms throughout the compilation process and at each point in the program the DSL author can know whether some value is a run-time value or not. We show the abstractions for tracking run-time-captured values in §14.3.

  Unlike with existing dynamic compilation systems the DSL compiler can be left without annotations. Its values are always static in user programs and therefore they do not affect dynamic compilation decisions.

- The `SD` abstractions are hidden from the DSL end-user with the type-driven translation. We explain how to modify the type translation of Yin-Yang to incorporate abstractions for dynamic compilation in §14.6.

- The DSL compiler tracks the places where values of type `SD` are used for making compilation decisions and collects the program slices that affect those values. The program slices are collected hidden by the `SD` abstraction. In our example, the compiler reifies and stores all computations on run-time-captured values in the unmodified dynamic programming algorithm [Cormen et al., 2001] for determining the optimal multiplication order (i.e., `c1 > c2`). Modifications to the DSL compiler framework to support program slicing are presented §14.4.

- Re-compilation guards are introduced automatically based on the reified decisions made during DSL compilation. In the example the re-compilation guard would be `c1 > c2`. Generation of re-compilation guards is presented in §14.7.

- Code for managing code caches is automatically generated based on the collected program slices. In the example the code cache would have two entries addressed with a single Boolean value computed with `c1 > c2`. Code cache management is presented in §14.8.

We evaluate 14.10 this approach on the matrix-chain multiplication algorithm described in 14.9. We show that the DSL end-user needs to add only 6 annotations in the original algorithm to enable automatic dynamic compilation. We further show that automatically introduced compilation guards reduce compilation overhead by 7.2x compared to best dynamic compilation approaches.

14.1 Approaches to Dynamic Compilation in DSLs

This section discusses different approaches to managing dynamic compilation of deeply embedded DSLs. The deep DSLs are specially hard for existing dynamic compilation systems as with those the whole DSL compiler would have to be annotated. Therefore, in this section, we discuss only the approaches that do not require annotation of the whole DSL compiler.The discussion is based on DSL compilers that use polymorphic embedding [Hofer et al., 2008]. In all examples we use Scala and for the abstraction of the IR of the DSL we use the type `R[_]`. Nevertheless, the discussion equally applies to different compilers and languages.

In type-directed partial evaluation [Danvy, 1999] and in LMS [Rompf and Odersky, 2012] staging is achieved by mixing host-language terms with DSL terms. The "Hello world!" example in such systems is the staged power function for natural numbers:

```
def pow(b: R[Int], e: Int): R[Int] =
  if (e == 0) 1
  else b * pow(b, e - 1)
```

In this example, since the exponent and the `pow` function is left as a host language term, execution of the function with statically known exponent performs partial evaluation. Calling this function is expanded into a chain of multiplications in the DSL IR:

```
pow(x, 3)
↪ x * x * x * 1
```

Recompilation on every execution. In LMS and type-directed partial evaluation there is no mechanism for managing re-compilation. The compilation happens on every execution and introduces large overhead in execution. For example, in Scala compiling the smallest program with a warmed-up compiler lasts more than 100 ms; due to large overhead we will not discuss this approach further.

14.1.1 Equality Guards

An approach to avoid re-compilation when input values are stable is to, for all captured values, memoize the value from the previous execution. Then, the re-compilation guards

can be introduced so they compare with the previous value. With this approach if the values are stable the compilation is avoided. This scheme is used in JIT compilers, DyC [Grant et al., 2000], and staging extensions for the Julia language [Bezanson et al., 2012].

This approach works well for programs that do simple partial evaluation (as the power function previously shown). In domain-specific optimizations, however, it is often the case that different input values produce the same output program (as in matrix-chain optimization). In these cases comparing for equality will introduce superfluous re-compilation.

Another problem with this approach is the notion of equality. In values passed by reference the dynamic system must choose between two types of equality: *i)* reference equality which is computed quickly but can cause unnecessary re-compilation when the object is equal only by value, and *ii)* value equality which is precise but can be very expensive to compute (e.g., in case of matrices).

14.1.2 Guards Based on IR Comparison

It is possible to avoid unnecessary re-compilation by deciding for re-compilation after optimizations are complete. With this approach static terms are executed in the host language and produce the final IR of the program. This IR can be used for comparing to previous executions (cache lookup) and deciding for re-compilation or re-use of previous programs.

With this approach re-compilation is precise—i.e., programs are re-compiled only if necessary—as the IR of the program exactly represents its semantics. If two IRs are the same than they can re-use the same compiled version of the code. This approach is only possible under the assumption that the compilation process is deterministic.

After the IR based cache lookup is performed, in the compilation pipeline, no further decisions can be made based on dynamic values. If a decision is made after the compilation process is not deterministic with respect to cache lookup. This would lead to semantic incorrectness of compilation guards.

The IR based cache lookup can be performed in any part of the compilation pipeline. For the purpose of DSLs the lookup can happen in two compilation phases: *i)* right after all optimization phases, and *ii)* lookup after code generation which compares the generated code of the programs.

Performance Overhead of IR Comparison. Although precise, IR comparison imposes un-necessary run-time overhead. Each time the DSL program is executed the

compilation pipeline until the cache lookup must be executed. Depending on the place of lookup in the compilation pipeline the overhead varies but they are always present. Performance of IR lookup is discussed in §14.10.

14.2 Program Slicing for Managing Dynamic Compilation

Managing dynamic compilation consists of making a decision whether to re-compile the code or reuse one of the previously compiled code versions. Since code caches are bounded, dynamic compilation management must introduce policies for managing the size of code caches.

To introduce precise and yet fast re-compilation decisions, it is necessary to execute only the subset of the compilation pipeline that leads to DSL compilation decisions that are based on run-time-captured values. This way, the dynamic compilation management is optimal as it does not introduce any run-time overhead.

A convenient technique for finding a subset of the whole program that affects a certain program value is *program slicing* [Weiser, 1981]. Program slicing is a technique for finding a subset of the program, i.e. a *program slice*), that affects a certain value, i.e., *slicing criterion*.

The main idea of this work is to track program slices of only run-time-captured values that affect compilation decisions, i.e. to use compilation decisions as slicing criteria. The program slices contain all the information necessary for re-computing the compilation decisions and therefore they can be used for introducing re-compilation logics.

Since static program slicing, is safely imprecise, it often includes large portions of the original program. To avoid this, in this work, we give complete control over program slicing to the DSL author. This way the DSL author can be sure that only the necessary parts of the compilation process are used for dynamic compilation management. To minimize development overhead of tracking program slices we introduce a type abstraction (`SD`), based on polymorphic embedding, that behind the scenes performs program slicing (§14.3).

14.3 Abstractions for Program Slicing

We introduce the abstract `type SD[+T]` that represents values captured at program run-time (run-time-captured values) that are used for making compilation decisions. In the compilation pipeline all run-time-captured values whose value changes over different executions and that affect compilation should be of type `SD`. For example, in a two-stage DSL a captured integer value should be of type `SD[Int]`.

We introduce an additional method `sd` for promoting run-time-captured values of type

`T` into the `SD[T]` type. This method is similar to the method `$lift` in the previously discussed deep embeddings. The difference is that it is used for lifting terms into a different domain. Signature of the `sd` method follows:

```
def sd[T](v: T): SD[T]
```

The function for returning from the `SD` domain to host language value is defined as:

```
def escape[T](v: SD[T]): T
```

In the context of DSLs the captured values should be converted to the `SD` type. Therefore, in DSLs that use this approach, the signature of the `$hole` method must be changed to:

```
def $hole[T](id: Int): SD[T]
```

The operations on type `SD` are defined with extension methods in the same way as operations are added to the `R` types in polymorphic embeddings. To introduce a new method, the DSL authors must introduce extension methods as described in §2.2. The framework described in this thesis, already provides the implementations for the base types of Scala.

Type `SD` is further refined with the interface for fetching its optional run-time-captured value and its optional static part:

```
trait SDOps[+T] {
    def static: Option[T]
    def dynamic: Option[R[T]]
  }
type SD[+T] <: SDOps[T]
```

Type `R` is the higher-kind abstract types used to hide the IR construction in polymorphic embeddings.

Values `static` and `dynamic` can be either *i)* both present, or *ii)* only one of them present. It should never happen that both the static part and the dynamic part of the expression are equal to `None`.

14.4 Reifying Dynamic Program Slices

To introduce optimal dynamic compilation management, it is necessary to have the IR of the trees that lead to compilation decisions. These trees can be used to generate re-compilation guards and cache management code that is executed before the DSL program. In this section we show how the `SD` type is used to perform execution and reification at the same time.

We define the `SD` as follows:

```
object SD {
  def apply[T](delayedS: => Option[T], delayedD: => Option[R[T]])
    : SD[T] = new SD(delayedS, delayedD)
}
class SD[+T](delayedS: => Option[T], delayedD: => Option[R[T]])
  extends SDOps[T] {
  lazy val static: Option[T] = delayedS
  lazy val dynamic: Option[R[T]] = delayedD
}
```

The type parameter `T` is declared as covariant so the types `SD[T]` behave equivalently to their type arguments with respect to subtyping. Variable `static` represents the optional run-time-captured values and the variable `dynamic` the optional reified trees. Here both `static` and `dynamic` are optional values as during compilation there are cases when it is not desired to have both the dynamic and static part of the term at the same time. Instances of `SD` are constructed with the `apply` method on the `object SD`.

Keeping track of both the dynamic and the static parts of the program is not new. Kenichi Asai [Asai, 2002] used this method for achieving partial evaluation in presence of data structures. Our approach differs as the values are explicitly tracked by the user and that both the static and the dynamic part of the term are optional.

Further, in the implementation, the `static` and `dynamic` values are lazy values, i.e., they are computed only once when accessed. The reason for delaying computation of the static and dynamic part is that values of type `SD` should not always be evaluated by value.

On the `SD` type we now define operations with extension methods. The method implementation performs both the execution and reification at the same time, given that the inputs are present (i.e., they are not `None`). For example an integer `+` operation on `SD[Int]` is defined as:

```
implicit class IntSD(lhs: SD[Int]) {
  def +(rhs: SD[Int]): SD[Int] = {
    SD(
      for (slhs <- lhs.static; srhs <- rhs.static)
        yield slhs + srhs,
      for (dlhs <- lhs.dynamic; drhs <- rhs.dynamic)
        yield dlhs + drhs
     )
  }
}
```

```
var reification = true
var execution = true

def withoutReification[T](b: =>T): T = {
  reification = false
  val res = b
  reification = true
  res
}

def withoutExecution[T](b: =>T): T = {
  execution = false
  val res = b
  execution = true
  res
}
```

Figure 14.1 – Dynamic scopes for disabling reification and execution in function bodies.

The `for` comprehensions used here, are equivalent to the `do` construct in Haskell. The result of the computation will be present only if both input parameters are of value `Some`. Otherwise, the result will be `None`.

14.4.1 Program Slicing with Functions

With the `SD` abstraction the function definitions can be either: *i)* left in the host language, or *ii)* abstracted over with `SD`. In the latter scenario it is necessary to provide operations for lifting functions in the `SD` domain and for function application. In the following examples, for simplicity, we leave functions in the host language. For example, an integer increment function can be defined as:

```
val inc: SD[Int => Int] = $lam((x: SD[Int]) => x + 1)
```

The `$lam` construct similarly to other constructs achieves both reification of the function and keeps the function value. Since execution of function application and its reification cause side-effects, function definitions must be such that during execution they do not perform reification of function bodies, and that during reification they do not perform execution of function bodies.

To allow blocking of all side-effects we define two global switches: one for disabling reification and one for disabling execution inside functions. They are controlled with dynamic scopes `withoutReification` and `withoutExecution` presented in Figure 14.1.

```
def $lam[T, U](f: SD[T] => SD[U]): SD[T => U] = SD(
  Some((x: T) =>
    withoutReification(f(SD(Some(x), None))).static.get),
  Some($lam((x: R[T]) =>
    withoutExecution(f(SD(None, Some(x)))).dynamic.get))
)

def $app[T, U](f: SD[T => U], v: SD[T]): SD[U] = SD(
  for (sf <- f.static; sv <- v.static) yield sf(sv),
  for (df <- f.dynamic; dv <- v.dynamic) yield $app(df, dv)
)
```

Figure 14.2 – Function definition and application with `SD` types.

For example, disabling reification and execution in the function `sd` is achieved as follows

```
def sd[T](v: T): SD[T] = SD(
  if(execution) Some(v) else None,
  if(reification) Some(unit(v)) else None
)
```

To achieve function execution without incurring re-reification of the function the `$lam` implementation uses the fact the `SD` arguments are optional and sets the dynamic scopes. For the `static` value of the function it uses η-expanded version of the input function with reification disabled and for reification the η-expanded version of the reification function with execution disabled. The implementation of the `$lam` function is shown in Figure 14.2. Note that methods `$app` and `$lam` are not recursive but they rather call into the methods of the same name that accept arguments of type `R`.

14.5 Tracking Relevant Program Slices

With explicit tracking of computations based on run-time-captured values it is easy to identify which run-time-captured values affect compilation decisions and to record their program slices. Not all slices of run-time-captured values affect compilation outcomes and only the relevant ones must be tracked. Consider the following snippet:

```
def toR[T](v: SD[T]): R[T] = v.dynamic
```

Here the run-time value of `v` is ignored and only its dynamic part is used. The run-time value of `v` does has no effect on compilation outcomes and therefore is not of interest for managing dynamic compilation.

Run-time value escape. The slices that are of interest are the ones where the `SD` value *escapes* into the host language. The value that escapes can be used to make decisions in

the compilation pipeline. For example, the *escaped value* can be used as the condition in the if statement that returns different IR nodes, or and index to a sequence that contains IR nodes.

The DSL author must mark escaped values explicitly by using a construct `escape` that converts and `SD` value into the host language value. The signature of `escape` is as follows:

```
def escape[T](v: SD[T]): T
```

When a value escapes the `SD` domain the program slice of that value is recorded. That information is than used for introducing optimal compilation guards §14.7 and code-cache management §14.8.

14.6 Concealing Dynamic Compilation Management

The presented abstractions are verbose and would hamper DSL development. For the DSL author we simplify programming with these abstractions with the use of implicit conversions and overloading resolution. For the DSL end-users we use Yin-Yang and introduce a new type translation for tracking run-time-captured values.

Concealing dynamic compilation from the DSL authors. To improve the interface for DSL authors we introduce several convenience methods that obviate manual annotation of code. The introduced simplifications are the following:

- **Implicit conversion of host language types.** Terms with host-language types are considered as always constant as they are not captured from user programs. Therefore, they can be freely promoted to the `SD` domain without user annotations. We define the function `sd[T](v: T): SD[T]` as implicit to minimize the number of user annotations.

- **Implicit conversion to the `R` type.** The `SD` typed values carry the program slice that computes them. To this end they can always be promoted to the domain of IR nodes (the `R` type). We make the function `toR[T]: SD[T] => R[T]` implicit. This means that the DSL author can drop the run-time-captured values without explicit annotations. This is safe, as dropping the run-time-captured values does not affect compilation decisions and, thus, does not affect dynamic compilation management.

- **Implicit function conversions.** We introduce implicit conversions for handing host language functions. With these conversions manual annotation of functions with `lam` and `app` is obviated in most of the cases.

- **Escape for conditionals.** If a condition of an `if` is of type `SD` and the branches are not (i.e., they are of type `R[T]` or `T`) the condition is treated specially. The condition of the `if` is treated as an escaped value, and the `if` construct is executed in the host language based on the run-time value of the conditional. We achieve this by using virtualization of `if`s and overloading the `$ifThenElse` function:

```
def $ifThenElse[T](c: SD[Boolean], t: => R[T], e: => R[T])
  : R[T] =
  if (escape(c)) t else e

def $ifThenElse[T](c: R[Boolean], t: => R[T], e: => R[T])
  : R[T] =
  IfThenElse(c, t, e)

def $ifThenElse[T](c: SD[Boolean], t: => T, e: => T): T =
  if (escape(c)) t else e

def $ifThenElse[T](c: SD[Boolean], t: SD[T], e: SD[T])
  : SD[T] = SD(
    c.static.flatMap(cs => {
      if (cs) t.static
      else e.static
    }),
    for(cd <- c.dynamic; td <- t.dynamic; ed <- e.dynamic)
      yield $ifThenElse(cd, td, ed)
  )
```

  For correct evaluation of conditionals it is necessary that `static` and `dynamic` parts are lazy evaluated. In case of `$ifThenElse` the static parts of a branch will only be executed if the condition for that branch is satisfied. Without making `static` and `dynamic` lazy it would not be possible to achieve this.

Concealing dynamic compilation from the DSL end-users. To hide the dynamic compilation abstractions from the users we introduce a translation from the direct embedding to the deep embedding. The translation, however, requires a different type translation as the deep embedding contains two type abstractions (`R` and `SD`).

To address this we introduce a type annotation in the user programs (`@static`) that is used to mark the types that should be translated into the `SD` type. These type annotations, however, appear in user programs only when the types must be stated explicitly. In most of the cases the type-checker in the deep embedding infers these types.

The translation rules for generic polymorphic embedding with dynamic compilation are

as follows:

$$\begin{aligned}\tau_{\texttt{arg}}(T) &= T\\ \tau(T) &= \texttt{R[}T\texttt{]}\\ \tau(T@static) &= \texttt{SD[}T\texttt{]}\end{aligned}$$

14.7 Compilation Guards

Based on reified program slices of escaped values we can introduce re-compilation guards that keep only a single version of the program. The, previously compiled, DSL program should be executed only if all escaped values are the same as in previous executions. The guard is then generated as all program slices of escaped values and an `if` statement whose condition is the conjunction of all escaped values that are re-computed. In the `if` statement, one branch contains the compiled DSL code and the other branch contains the re-compilation trigger.

We demonstrate generated guards on a two-stage sign function:

```
def sgn(v:  Int): Int = dsl {
  if (v > 0) math.POSTIVE
  else if (v < 0) math.NEGATIVE
  else math.ZERO
}
```

In this example the escaped values are the two conditions that are passed to `if` statements. They are affecting the resulting IR of the compilation process. After translation the DSL body is translated into

```
 $ifThenElse(hole(1, $tpe[Int]) > $lift(0),
   $lift(math).POSITIVE,
   $ifThenElse(hole(1, $tpe[Int]) < $lift(0),
     $lift(math).NEGATIVE, $lift(math).ZERO)
)
```

where `$lift` is semantically equivalent to `sd` with a signature that matches the translation.

The overloading resolution resolves the `ifThenElse` function calls to ones that mark the condition as escaped. Due the escapes there are two program slices that affect compilation decisions: *i)* `v > 0` and *ii)* `v < 0`. For example, the `sgn` function called with the argument `0` results in[1]:

[1] The wrapper code that leads to the compilation guards (e.g., the `execute` method) is omitted for simplicity.

```
sgn(0)
↪
if (v > 0 == false && v < 0 == false)
  math.ZERO // optimized program
else
  recompileAndExecute(v) // re-compilation and re-execution.
```

The example presented here does not improve performance of the original program. The `if` statements, that are removed based on the knowledge of the value of `v`, are re-introduced as guards in front of the block. In other examples, however, the removal of conditionals opens possibilities for further optimizations and can lead to large improvement in performance. This is especially applicable to domain-specific optimizations as they can yield orders of magnitude improvements in performance.

14.8 Code Cache Management

For introducing re-compilation guards the order of execution of captured program slices is not important. There is only one version of the generated code and if any of the escaped values does not match previous executions the program is re-compiled. Re-compilation introduces a new guard statement that guards the new generated code.

Keeping multiple versions of the generated code, i.e. having code caches, requires more than using a simple conjunction of program slices. Since we track paths to escaped values, for code caches:

- There is an exact order in which values escape.
- Not all value escapes are known in a given point of time.
- The code caches are always bounded by the DSL author or machine resources. Therefore it is necessary to decide on eviction policies based on usage frequency.

To address these requirements for code caches we persist program slices of escaped values over different program compilations, track dependencies between different escaped values, and track information about the number executions for each cached value.

14.8.1 Persistence and Evolution of Program Slices

To track escaped values and their program slices over consecutive executions we define a tree structure. The tree is defined by following abstract data types:

```
trait Node
case class Escape(slice: R[Any], val children: Map[Any, Node])
```

```
    extends Node
case class Leaf(var count: Long) extends Node
```

The node `Escape` represents each escaped value in the program. It carries the value `slice` that represents the program slice that affects that escaped value and the `children` nodes. The children represent the following escaped values on the execution path or the terminal node `Leaf`. Leafs represent the end of compilation and carry a single variable `count` that is used for tracking execution frequency of the given paths.

The tree of escaped values is stored and maintained over consecutive executions of the program. The tree is maintained in two ways:

- On re-compilation the compiler will reach a path that has not been explored. This expands that branch of the tree if new value escapes happen.
- On successful execution only the `count` of the corresponding path is incremented.

14.8.2 Generating Code Caches

For each node in the tree the framework generates a part of re-compilation management code. The algorithm is structurally recursive on the tree structure:

`Escape` nodes. For each `Escape` node the algorithm generates a multi-way branch that branches to all of the children based on their values. In case of escaped `Boolean` values the `if` statement is generated instead of the pattern match[2].

For example, a simple slice that accepts a run-time value (e.g., variable `x`) and that escapes, and in previous executions it had values 1, 4, and 2 the generated statement is as follows:

```
x match {
  case 1 => // code generated by recursive calls
  case 4 => // code generated by recursive calls
  case 2 => // code generated by recursive calls
}
```

In case none of the children lead to entries for which there is code in the cache for the `Escape` node the generated code only calls into re-compilation:

```
recompileAndExecute(arguments)
```

[2]In Scala the `switch` statement is expressed with pattern matching

`Leaf` nodes. The base cases in recursion are the `Leaf` nodes. For the leaf nodes there are two cases for code generation:

- When the leaf node is in the code cache and should stay there the algorithm generates:

  ```
  execute(Array[Byte](1b))(args)
  ```

  The byte array represents all the concrete values on the path to this node. This array is used for indexing the code cache for the correct code version. In this byte array, for performance reasons, Boolean values are encoded as bits. Arguments `args` is a sequence of all run-time-captured values.

  The function `execute` is implemented to simply look for a function that accepts the `args` in the cache based on the path (the byte array) and execute that function.

- When the leaf node is not in the cache or should be evicted based on the LRU policies the following call is generated:

  ```
  recompileAndExecute(arguments)
  ```

  Function `recompileAndExecute` calls recompilation that will further evolve the tree of escaped values and generate a new guard statement.

14.8.3 Example of Code Caches for the Sign Function

Here we present how code caches are managed on the example of the `sgn` function defined previously. The code in the graphs is based on the body of the `sgn` function. Further, this example does not introduce speedups for the given function, but it is merely used to describe the technique. With the presented technique speedups are noticeable with domain-specific optimizations §14.9.

We assume that the code cache size for this function can contain at most two entries. Then we show how we track execution and the generated code caches for the following sequence of `sgn` executions:

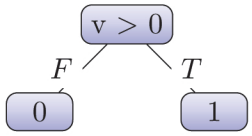
```
sgn(1); sgn(0); sgn(0); sgn(-1);
```

In the following figures on the left side we present the state of the tree that tracks escaped values. The edge labels represent the last value of the escaped code. The F sign stands for false and the T sign for true. The right side shows the code that performs re-compilation management for the given state of the tree.

Before the first execution of the `sgn` function the tree is empty and the guard always calls into re-compilation:

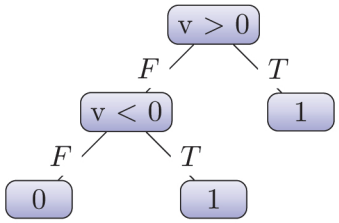

```
recompileAndExecute(v)
```

Then, after the first execution with `sgn(1)`, the code is re-compiled and the tree is augmented with new paths. The guard now reuses the old code for values that are greater than 0:



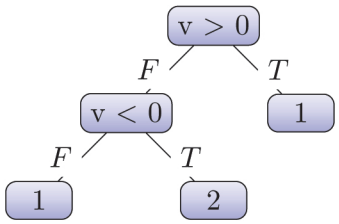

```
if (v > 0)
  execute(Array[Byte](1b))(v)
else
  recompileAndExecute(v)
```

At this point there is only one value in the code cache. When the function is executed again with `sgn(0)` the re-compilation occurs and the tree is augmented with the new path:




```
if (v > 0)
  execute(Array[Byte](1b))(v)
else if (v < 0)
  recompileAndExecute(v)
else
  execute(Array[Byte](00b))(v)
```

The consecutive execution simply augments the count of executions for the case when `v == 0`. Now, when the `sgn(-1)` is executed the re-compilation is executed again. Since the code cache is full the compiler must evict the previously compiled values. Since, code the case when `v > 0` is executed only once that value is evicted from the cache. The number of executions for that branch, however, is preserved for future decisions. The state after calling `sgn(-1)` follows:




```
if (v > 0)
  recompileAndExecute(v)
else if (v < 0)
  execute(Array[Byte](01b))(v)
else
  execute(Array[Byte](00b))(v)
```

14.9 Case Study: Matrix-Chain Multiplication

In a chain of matrix multiplications the order in which multiplications are executed affects the program execution-time in orders of magnitude. For choosing the right order

of multiplications it is necessary to estimate the cost of execution based on matrix sizes which are known only at runtime.

In this section we show the algorithm for choosing the right order of matrix multiplications. Then we explain the steps that the DSL author needs to perform in order to convert the algorithm to the version that uses the `SD` abstraction. In §14.10 we evaluate the performance of the code generated by the algorithm defined here.

The basic algorithm for deciding on the optimal order of multiplications is based on dynamic programming. The algorithm is executed on two matrices that contain the information about the cost of multiplication matrix sub-chains. For all the following example the matrices will be in scope with the following definition:

```
var m: Array[Array[Int]] = _
var s: Array[Array[Int]] = _
```

To adapt this algorithm for dynamic compilation the types of matrices need to be changed to `SD[Array[Array[Int]]]`. All of the costs are computed based on the run-time-captured values so the matrices that hold them should be as well:

```
var m: SD[Array[Array[Int]]] = _
var s: SD[Array[Array[Int]]] = _
```

An alternative approach would be to define types of `m` and `s` as `Array[Array[SD[Int]]]`. With this approach all operations on the `m` and `s` would be executed in the host language. For large matrix chains this can lead to code explosion in guards so we choose the type `SD[Array[Array[Int]]]`.

The algorithm that defines the order of matrix multiplications is presented in Figure 14.3. The algorithm consists of two functions:

- Function `cost` that is used to populate the matrices with costs of computation for different sub-chains.
- Function `optimalChain` that is used to take an original order and produce the new order based on the costs from the `m` matrix.

In the modified version of the algorithm we can see how the DSL author defines the program slice that is used for compilation decisions based on run-time-captured values:

- In the `cost` function the argument `p` that carries the array of matrix sizes is changed to a run-time value with `p: SD[Array[Int]]`. Then the cost matrices `m` and `s` are initialized by lifting the `Array` object with `sd`:

```
def cost(p: Array[Int]) {
  val n = p.length - 1
  m = Array.fill(n, n)(0)
  s = Array.fill(n, n)(0)
  for (ii <- 1 until n;
       i  <- 0 until n - ii) {
    val j = i + ii
    m(i)(j) = Int.MaxValue
    for (k <- i until j) {
      val q = m(i)(k) +
        m(k + 1)(j) +
        p(i) * p(k+1) * p(j+1)
      if (q < m(i)(j)) {
        m(i)(j) = q
        s(i)(j) = k
      }
    }
  }
}

def optimalChain(
  m: Array[Exp[Matrix[Double]]])
  : Exp[Matrix[Double]] = {
  def chain(i: Int, j: Int)
    : Exp[Matrix[Double]] =
    if (i != j) MatrixMult(
      chain(i, s(i)(j)),
      chain((s(i)(j) + 1), j))
    else m(i)

  chain(0, s.length-1)
}
```

```
def cost(p: SD[Array[Int]]) {
  val n = p.length - 1
  m = sd(Array).fill(n, n)(0)
  s = sd(Array).fill(n, n)(0)
  for (ii <- 1 until n;
       i <- 0 until n - ii) {
    val j = i + ii
    m(i)(j) = Int.MaxValue
    for (k <- i until j) {
      val q = m(i)(k) +
        m(k + 1)(j) +
        p(i) * p(k+1) * p(j+1)
      if (q < m(i)(j)) {
        m(i)(j) = q
        s(i)(j) = k
      }
    }
  }
}

def optimalChain(
  m: Array[Exp[Matrix[Double]]])
  : Exp[Matrix[Double]] = {
  def chain(i:SD[Int], j:SD[Int])
    : Exp[Matrix[Double]] =
    if (i != j) MatrixMult(
      chain(i, s(i)(j)),
      chain((s(i)(j) + 1), j))
    else m(escape(i))

  chain(0, s.length-1)
}
```

Figure 14.3 – Basic algorithm for deciding the optimal order of matrix-chain multiplication. The left side displays the original algorithm, and the right side the modified version of the algorithm.

```
m = sd(Array).fill[Int](n, n)(0)
s = sd(Array).fill[Int](n, n)(0)
```

Here, the constant `0` is promoted with implicit conversions to a dynamic value based on the signature of the lifted `Array.type` (`SD[Array.type]`). In user code most of the constants are promoted in the same way. The rest of the function `cost` is promoted to `SD` types either by implicit conversions or the overloading resolution on common methods.

The `optimalChain` function makes decisions on the order of multiplications based on the costs collected in `cost` function. In `optimalChain` the author must re-define the recursive function `chain0` to operate on `SD` values as those are the values based on which the decisions are made. In this function, the places where the run-time-captured values escape to the host language are: *i)* the `if` statement that accepts `SD[Boolean]` as the condition but returns the IR of the DSL and *ii)* the array indexing operation `m(escape(i)`. Note that once array indexing happens all decisions are already made so there will be no additional guards after that point.

14.10 Evaluation

This section evaluates dynamic compilation management introduced in this thesis. The evaluation is performed on the matrix-chain multiplication algorithm defined in the previous section (§14.9). Evaluation covers execution time of the guard statements (§14.10.1) and verifies that the size of generated code is small enough to be efficiently executed on a target platform (§14.10.2). All experiments are performed on a simple program that contains only the matrix-chain.

We ran 6 different matrix multiplication chains with sizes from 3 to 8 matrices. The input matrices are chosen such that they always generate new cache entries. The algorithm is run with the 1, 5, 10, and 20 paths that are explored (cache sizes). For chains of size 3, 4, and 5, some cache sizes are displayed as there is more cache entries than multiplication orders.

14.10.1 Matrix-Chain Multiplication: Execution Time

To measure solely the execution time of guard statements, we changed the implementation of guard generation. In the modified code, the time measure is taken before the guard starts executing and in each leaf branch of a guard statement. Two measure are compared to determine the guard execution time in isolation.

Figure 14.4 displays execution times for different matrix chain lengths and different number of paths in a guard. The execution times of guards in all cases are smaller

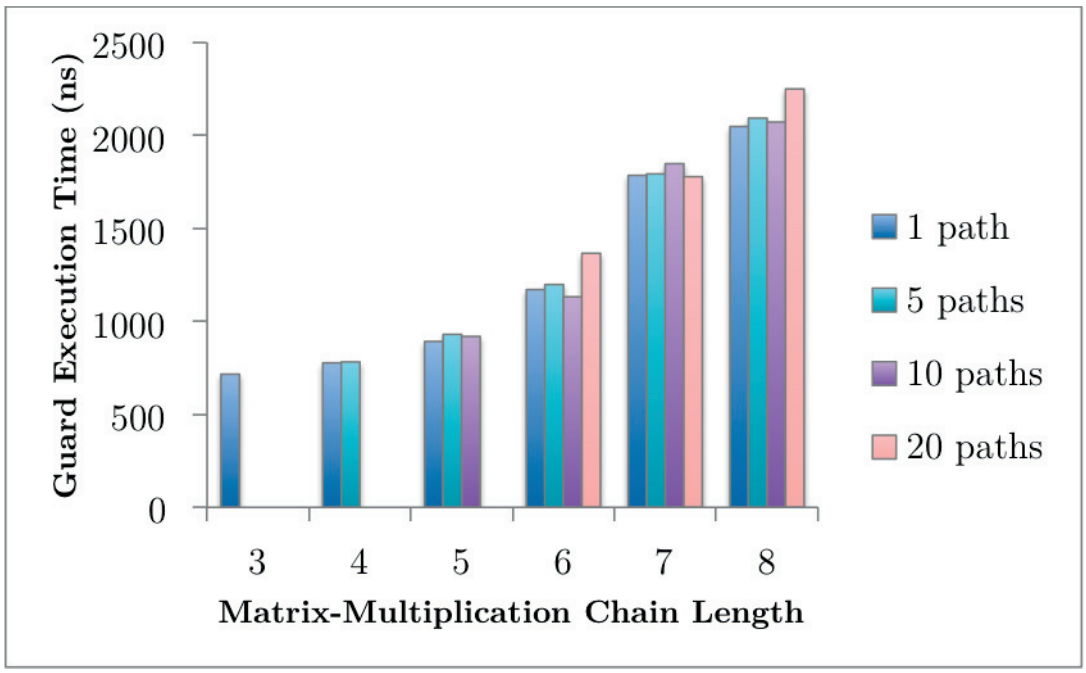


Figure 14.4 – Execution time of re-compilation guards for matrix-chain multiplication. The x-axis represents the length of the multiplication chain, the y-axis represents the execution time in nanoseconds, and different bars represent the number of explored paths (i.e., the number of slots in the code cache).

than 2.5 μs. The execution time of the guards grows quadratically with the number of matrices in the chain. The reason for this growth is the quadratic nature of the chosen algorithm for computing the optimal chain and use the standard benchmarking methodology (§A.2).

Table 14.1 – Comparison of dynamic compilation based on IR comparison with dynamic compilation based on program slicing and guard generation.

Matrix-chain length	IR comparison (μs)	Slicing (μs)	Speedup
3	723	0.542	1333.9
4	996	0.749	1329.8
5	1259	0.909	1385
6	1587	1.366	1161.8
7	1861	1.778	1046.7
8	2134	2.25	948.4

Since, to our knowledge, no DSLs do dynamic compilation management we introduced this technique to LMS. We introduced dynamic compilation management based on IR comparison, that compares the IR of the current execution with the previous IRs after all optimization phases. We do not measure executions for different cache sizes as the performance of the algorithm only marginally depends on the size of the cache.

Table 14.1 displays the comparison. Dynamic compilation management based on program slices always outperforms the IR comparison method. Execution time of guards with largest code cache is in all cases 3 orders of magnitude smaller than the IR comparison.

In cases when the DSL programs are larger the improvements in cache lookup time can be bigger, depending on the program size. What is presented here is the best case for the technique of IR comparison as the programs do not have IR nodes that are unrelated to dynamic compilation. These nodes would further increase the time required for IR comparison.

14.10.2 Matrix-Chain Multiplication: Byte-Code Size

On the matrix-chain multiplication benchmark we evaluated if the guards are small enough so they can be efficiently executed on the JVM. The guard size must not exceed the maximum size of methods on the JVM and must be small enough to be JIT compiled.

For all guards in our benchmarks the byte-code size is significantly smaller than the limits that the JVM imposes. For all guards in our benchmarks the guards are small enough to be always JIT compiled.

14.11 Related Work

DyC [Grant et al., 2000] is an annotation-based dynamic compilation system for C. In DyC programmers can annotate variables for which dynamic compilation is applied as well as declare different cache policies for relevant program points. The set of caching policies in DyC is greater than what is presented in this thesis. The drawback of DyC is its rudimentary support for partially-static data structures. To avoid run-time overheads users are required to annotate pointer dereferences for partially-static data.

In case of DSLs, the intermediate representation is partially static. This means that with DyC, it is not possible to write DSLs without additional user annotations that would impair end-user experience.

Truffle [Würthinger et al., 2013] is a framework for dynamic compilation of language interpreters. In Truffle, DSL authors have precise control over code caches, assumptions related to run-time variables, and code specialization. Although Truffle is a weapon of choice for dynamic languages, in presence of statically compiled DSLs it has shortcomings:

- Truffle requires running the programs first, before performing optimizations. This is not well suited for global program transformations as parts of the program that should be transformed by global optimizations can be already executed.
- If used in combination with a static compiler, Truffle requires DSL authors to

declare code caches and assumptions imperatively. There are not abstractions, similar to `SD`, that would hide this from the DSL author.

Truffle could be effectively used as the back-end for dynamic compilation based on program slicing. With Truffle, the generated guards would be tightly integrated with the JIT compiler yielding better performance and smaller code.

A Appendix

A.1 Yin-Yang Translation Correctness

In this section we formalize the core translation of Yin-Yang in rank-1 polymorphic lambda-calculus with the **let** construct for introducing universally quantified terms. To the core calculus we add primitive types **Bool** and **Int**. The calculus does not include type reconstruction: all type annotations are explicitly stated. The syntax of our core language is the following:

Terms:

$$t ::= t\ t \mid x[\tau] \cdots [\tau] \mid \textbf{let}\, x : \sigma = t\, \textbf{in}\ t \mid v$$
$$v ::= \lambda x : \tau.t \mid c$$
$$c ::= \textbf{true} \mid \textbf{false} \mid 0 \mid 1 \mid ...$$

Types:

$$\sigma ::= \tau \mid \forall X.\sigma$$
$$\tau ::= X \mid \tau \Rightarrow \tau \mid \iota$$
$$\iota ::= \textbf{Bool} \mid \textbf{Int}$$

Since the evaluation rules for this calculus are well known we state only the typing rules in Figure A.1.

To simulate the deep embedding, and at the same time keep the calculus of the meta-language simple, we introduce additional constructs in the object language. We assume that the object language supports type-application at the type level for the deep embedding type `R`. The `R` can be viewed as a built-in higher-kind type (similar to `List` or `Array`). For all τ types $R[\tau_1] = R[\tau_2]$ if and only if $\tau_1 = \tau_2$. The type substitution for `R` types behaves as expected. Terms of type `R` are instantiated with the `lift` function that is always in context in the object language. The `lift` function has the following type in

Type Rules

$$\Gamma \vdash \mathbf{true}, \mathbf{false} : \mathbf{Bool}\ (\text{T-Bool}) \qquad \Gamma \vdash 0, 1, ... : \mathbf{Int}\ (\text{T-Int})$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . t : \tau_1 \Rightarrow \tau_2}\ (\text{T-Abs}) \qquad \frac{\Gamma \vdash t_1 : \tau_1 \Rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}\ (\text{T-App})$$

$$\frac{x : \forall X_1. \cdots \forall X_n . \tau \in \Gamma}{\Gamma \vdash x[\tau_1] \cdots [\tau_n] : [X_1/\tau_1] \cdots [X_n/\tau_n]\tau}\ (\text{T-TApp})$$

$$\frac{\Gamma \vdash t_1 : \sigma \quad \Gamma, x : \sigma \vdash t_2 : \tau}{\Gamma \vdash \mathbf{let}\, x : \sigma = t_1\, \mathbf{in}\ t_2 : \tau}\ (\text{T-Let})$$

Figure A.1 – The Typing Rules.

context:

$$\Gamma \vdash lift : \forall X. X \Rightarrow R[X]$$

For types, we define the equivalence relation (=) structurally:

$$\mathbf{Bool} = \mathbf{Bool} \qquad \mathbf{Int} = \mathbf{Int} \qquad X = X$$

$$\frac{\tau_1 = \tau_1' \quad \tau_2 = \tau_2'}{\tau_1 \Rightarrow \tau_2 = \tau_1' \Rightarrow \tau_2'} \qquad \frac{\tau = \tau'}{\forall X. \tau = \forall X. \tau'}$$

The described equivalence relation requires type variables to have the same name for the types to be equivalent. This, "stricter" version of the equivalence relation is chosen as it makes proving the following theorem simpler.

Theorem. *If a term $\Gamma \vdash t : \sigma$ is well typed and the translation is successful, the translated term's type is equivalent to the type translation of the original term's type $\Gamma \vdash [\![t]\!] : \langle \tau \rangle$.*

Proof. We conduct the proof by using induction on the term translation ($[\![\_]\!]$):

Case $[\![\_]\!]$-Const **:** For translation of constants there are two cases in the typing derivation:

- *Case* T-Bool **:** Based on the premise the type of a translated term is **Bool**. Given the signature of $lift$, the translated term has a type $R[\mathbf{Bool}]$. This is equivalent to the type translation of the original term's type ($\langle \_ \rangle$-Base).
- *Case* T-Int **:** Analogous to T-Bool.

Term Translation ($[\![\_]\!]$)

$$[\![\lambda x : \tau.t]\!] = \lambda x : \langle\tau\rangle.[\![t]\!] \; ([\![\_]\!]\text{-Abs}) \qquad [\![t_1 \; t_2]\!] = [\![t_1]\!] \; [\![t_2]\!] \; ([\![\_]\!]\text{-App})$$

$$\frac{\Gamma \vdash c : \iota}{[\![c]\!] = \texttt{lift}[\iota] \; c} \; ([\![\_]\!]\text{-Const})$$

$$[\![x[\tau_1] \cdots [\tau_n]]\!] = x[\langle\tau_1\rangle_{arg}] \cdots [\langle\tau_n\rangle_{arg}] \; ([\![\_]\!]\text{-TApp})$$

$$[\![\textbf{let}\, x : \sigma = t_1 \,\textbf{in}\, t_2]\!] = \textbf{let}\, x : \langle\sigma\rangle = [\![t_1]\!] \,\textbf{in}\, [\![t_2]\!] \; ([\![\_]\!]\text{-Let})$$

Type Translation ($\langle\_\rangle$ and $\langle\_\rangle_{arg}$)

$$\langle\iota\rangle = R[\iota] \; (\langle\_\rangle\text{-Base}) \qquad \langle X\rangle = R[X] \; (\langle\_\rangle\text{-TVar})$$

$$\langle\tau_1 \Rightarrow \tau_2\rangle = \langle\tau_1\rangle \Rightarrow \langle\tau_2\rangle \; (\langle\_\rangle\text{-Func}) \qquad \langle\forall X.\sigma\rangle = \forall X.\langle\sigma\rangle \; (\langle\_\rangle\text{-Abs})$$

$$\langle\tau_1 \Rightarrow \tau_2\rangle_{arg} = \mathit{error} \; (\langle\_\rangle\text{-FArg}) \qquad \frac{\tau \neq \tau_1 \Rightarrow \tau_2}{\langle\tau\rangle_{arg} = \tau} \; (\langle\_\rangle\text{-Arg})$$

Figure A.2 – Yin-Yang Translation.

Case $[\![\_]\!]$-Abs **:** By applying the typing derivation T-Abs on the translated term and the induction hypothesis on the abstraction body, the translated term's type is $\langle\tau_1\rangle \Rightarrow \langle\tau_2\rangle$. This is equivalent to the type translation of the original term's type ($\langle\_\rangle$-Func).

Case $[\![\_]\!]$-App **:** By applying the induction hypothesis on both sub-terms of the application and applying the typing rule T-App, the type of the translated term is $\langle\tau_2\rangle$. This is equivalent to the type translation of the original term's type.

Case $[\![\_]\!]$-TApp **:** There are two cases to consider for this translation rule:

- When one of the type arguments is a function type the translation rule $\langle\_\rangle$-FArg is triggered and the type translation is not successful. The theorem holds as the translated term was not well formed.
- By applying the typing derivation on the translated term and comparing it to the type translation of the original term the following needs to hold:

$$\langle[X_1/\tau_1] \cdots [X_n/\tau_n]\tau\rangle = [X_1/\langle\tau_1\rangle_{arg}] \cdots [X_n/\langle\tau_n\rangle_{arg}]\langle\tau\rangle$$

  This equality holds by induction on the type structure:

  Case Base type **:** Trivially holds.

  Case Type variable **:** Holds as the substituted type must be either a base type

or a type variable. For both cases it trivially holds by following translation rules $\langle\_\rangle$-Base and $\langle\_\rangle$-TVar, and type substitution.

Case Function type **:** Holds by the induction hypothesis, substitution, and type translation rules for functions.

Case $[\![\_]\!]$-Let **:** By the induction hypothesis on sub-terms in the translation and the T-Let rule.

□

A.2 Hardware and Software Platform for Benchmarks

In this thesis all benchmarks are executed on the same platform:

- **Hardware configuration** is the following: the *CPU* is Intel Core i7-2600K with 4 physical cores each having 2 virtual-cores and has 3.4 GHz working frequency, the *main memory* is DDR3 with 1333 MHz working frequency. We assure that hyper-threading and frequency-scaling are disabled during executions.
- **Software configuration** is the following: for compilation of Scala programs we use Scala 2.11.7, the programs execute on the HotSpot 64-Bit Server (24.51-b03) virtual machine.

Measurements are performed in the same way for all benchmarks. Each benchmark is executed until the virtual machine is warmed up and stabilized. During the measurements we assure that no garbage collection happens. Finally, all reported numbers in the these are a mean of the last 10 measurements. In all experiments the variance is smaller than 5%.

Bibliography

Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):237–268, 1991.

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F Sweeney. Adaptive optimization in the Jalapeno JVM. In *International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2000.

Kenichi Asai. Binding-time analysis for both static and dynamic expressions. *New Generation Computing*, 20(1):27–51, 2002.

Joel Auslander, Matthai Philipose, Craig Chambers, Susan J Eggers, and Brian N Bershad. Fast, effective dynamic compilation. In *International Conference on Programming Language Design and Implementation (PLDI)*, 1996.

Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

Edwin C Brady and Kevin Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *International Conference on Functional Programming (ICFP)*, 2010.

K. J Brown, A. K Sujeeth, H. J Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT)*, 2011.

Eugene Burmako. Scala macros: Let our powers combine! In *Workshop on Scala (SCALA)*, 2013.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

H. Chafi, Z. DeVito, A. Moors, T. Rompf, A.K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In

Proceedings of the ACM international conference on Object oriented programming systems languages and applications, page 835–847, 2010.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *International Conference on Functional Programming (ICFP)*, 2007.

Krzysztof Czarnecki, John O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. *Domain-Specific Program Generation*, 2004.

Olivier Danvy. *Type-directed partial evaluation*. Springer, 1999.

Conal Elliott, Sigbjørn Finne, and Oege De Moor. Compiling embedded languages. *Journal of functional programming*, 13(03):455–481, 2003.

Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *ECOOP 2007–Object-Oriented Programming*, pages 273–298. Springer, 2007.

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. In *International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2011.

Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *Software language engineering*, pages 197–217. 2013.

Martin Fowler. Language workbenches: The killer-app for domain specific languages. 2005.

Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing Kansas Lava. In *Symposium on Implementation and Application of Functional Languages (IFL)*. 2011.

Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1):147–199, 2000.

Danny M Groenewegen, Zef Hemel, Lennart CL Kats, and Eelco Visser. WebDSL: a domain-specific language for dynamic web applications. In *Companion to the International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA Companion)*, 2008.

Miguel Guerrero, Edward Pizzi, Robert Rosenbaum, Kedar Swadi, and Walid Taha. Implementing DSLs in metaOCaml. In *International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2004.

Jurriaan Hage and Bastiaan Heeren. Heuristics for type error discovery and recovery. In *Symposium on Implementation and Application of Functional Languages (IFL)*. 2007.

Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

Bastiaan Heeren, Jurriaan Hage, and S Doaitse Swierstra. Scripting the type inference process. In *International Conference on Functional Programming (ICFP)*, 2003.

Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In *Programming Languages and Systems (ESOP)*, 1994.

C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *International Conference on Generative Programming and Component Engineering (GPCE)*, 2008.

Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, 1996.

Paul Hudak. Modular domain specific languages and tools. In *International Conference on Software Reuse (ICSR)*, 1998.

Lennart CL Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2010.

Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2009.

Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. 2014.

Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LISP and functional programming*, 1986.

Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *International Conference on Software Language Engineering (SLE)*, pages 311–331. 2013.

Prasad A. Kulkarni. JIT compilation policy for modern machines. In *International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011.

Anne-Françoise Le Meur, Julia L Lawall, and Charles Consel. Specialization scenarios: A pragmatic approach to declaring program specialization. *Higher-Order and Symbolic Computation*, 17(1-2):47–92, 2004.

HyoukJoong Lee, Kevin J Brown, Arvind K. Sujeeth, Hassan Chafi, Kunle Olukotun, Tirark Rompf, and Martin Odersky. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, (5):42–53, 2011.

Tom Lokhorst. Awesome prelude, 2012. Dutch Haskell User Group, http://vimeo.com/9351844.

Russell McClure, Ingolf H Krüger, et al. SQL DOM: compile time checking of dynamic sql statements. In *International Conference on Software Engineering (ICSE)*, 2005.

Heather Miller, Philipp Haller, and Martin Odersky. Spores: a type-based foundation for closures in the age of concurrency and distribution. In *European Conference on Object-Oriented Programming (ECOOP)*. 2014.

A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, page 117–120, 2012.

Tridib Mukherjee, Ayan Banerjee, Georgios Varsamopoulos, Sandeep KS Gupta, and Sanjay Rungta. Spatio-temporal thermal-aware job scheduling to minimize energy consumption in virtualized heterogeneous data centers. *Computer Networks*, 53(17): 2888–2904, 2009.

Vladimir Nikolaev. Sprinter. http://vladimirnik.github.io/sprinter/.

M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. *The Scala Language Specification (Version 2.9)*. 2014.

Martin Odersky. Contracts for scala. In *Runtime Verification*, pages 51–57, 2010.

Martin Odersky and Matthias Zenger. Scalable component abstractions. In *International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2005.

Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2010.

Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *ACM SIGPLAN Notices*, volume 23, page 199–208, 1988.

Markus Püschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gačic, Yevgen Voronenko, et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2): 232–275, 2005.

T. Rompf, A.K. Sujeeth, H.J. Lee, K.J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented DSLs. In *IFIP Working Conference on Domain-Specific Languages (DSL)*, 2011.

Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6): 121–130, 2012.

Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-virtualized: Linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, page 1–43, 2013a.

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanović, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Symposium on Principles of Programming Languages (POPL)*, 2013b.

Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Kunle Olukotun, and Martin Odersky. Project Lancet: Surgical precision JIT compilers. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2013c.

Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2012.

Maximilian Scherr and Shigeru Chiba. Implicit staging of EDSL expressions: A bridge between shallow and deep embedding. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014.

P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *International conference on Management of data (SIGMOD)*, pages 23–34, 1979.

A. Shaikhha and M. Odersky. An embedded query language in Scala. Technical report, Technical Report EPFL-STUDENT-213124, EPFL, Lausanne, Switzerland, 2013.

Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An implicitly parallel domain-specific language for machine learning. In *International Conference on Machine Learning (ICML)*, 2011.

Arvind K. Sujeeth, Austin Gibbons, Kevin J Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: generating a high performance DSL implementation from a declarative specification. In *International Conference on Generative Programming and Component Engineering (GPCE)*, 2013a.

Arvind K. Sujeeth, Tiark Rompf, Kevin Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksander Prokopec, Vojin Jovanović, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *European Conference on Object-Oriented Programming (ECOOP)*, 2013b.

Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for EDSL. In *Trends in Functional Programming (TFP)*, pages 21–36, 2013.

Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 1997.

Typesafe. Slick. http://slick.typesafe.com/.

Vlad Ureche, Tiark Rompf, Arvind K. Sujeeth, Hassan Chafi, and Martin Odersky. StagedSAC: a case study in performance-oriented DSL development. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2012.

Tijs van der Storm. *The Rascal language workbench*. CWI. Software Engineering [SEN], 2011.

Eelco Visser. Program transformation with Stratego/XT. In *Domain-Specific Program Generation*. 2004.

Mark Weiser. Program slicing. In *International conference on Software engineering (ICSE)*, 1981.

Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, 2013.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

Curriculum Vitae

Personal Information

Full Name	Vojin Jovanovic
Address	Goldbrunnenstrasse 73, 8055, Zürich, Switzerland
Telephone	+41 (0)21 69 37691 Mobile: +41 (0)78 871 91 74
Email	vojin.jovanovic@epfl.ch
Date of birth	14th January 1985

Goals and aspirations: I believe that programs can be written abstractly and yet execute as fast as their hand tuned counterparts. To this end, I am making a framework that allows effortless addition of domain-specific optimizations to existing libraries. I am also working on a high-level programming model for dynamic compilation where dynamic information is used to perform domain-specific optimizations at runtime; yet managing assumptions, deoptimization, and code caches is done behind the scenes.

Selected Work Experience

Position and Dates: **PhD Student** October 2010 – December 2015

Employer: Scala Laboratory (LAMP), EPFL, Switzerland

Main activities and responsibilities:
Author and maintainer of the Yin-yang framework which is used for seamless embedding of DSLs. Yin-yang is used for generating and reifying queries in the new version of LegoBase.
Co-author and initiator of Scala Records. Scala Records are used for type-safe manipulation of SparkSQL query results.
Co-author of SIP 14 – Futures and Promises
LMS contributor: implemented a loop fusion prototype, added record support, enabled and helped removal of the dependency to Virtualized Scala.
Author of sbt-coursera which is used for automatic grading of Java based projects Coursera.
Co-author of the Actors Migration Kit.

Position and Dates: **Research Intern** June 2013 – September 2013

Employer: Oracle Labs, Switzerland

Main activities and responsibilities: Implemented the Graal backend for Lightweight Modular Staging with support for vectorization. Performance on all (at the time) supported vectorization features was within 10% of hand-written C.

Position and Dates: **Research Intern** April 2010 – September 2010

Employer: Network Systems Laboratory (NSL), EPFL, Switzerland

Main activities and responsibilities: Implemented DiCE, a system that makes a snapshot of a network of BGP routers and uses concolic execution to explore the live system state. Exploration ensures that the faulty system states can not be reached. DiCE detects common errors in BGP networks like the cybernuke vulnerability.

Position and Dates: **Software Developer – Team Leader** March 2008 – September 2009

Employer: Margintech Corporation, Toronto
Working for Taleo inc. on the TBE product

Main activities and responsibilities: Developed software for a large SaaS system that is used daily by over 50.000 customers. Lead a team of 3 people on several enterprise projects. At the same time developed algorithms for cache re-balancing (10x improvement in memory utilization), fixed critical concurrency bugs and integrated a semantic search engine.

Education

School and Dates: École polytechnique fédérale de Lausanne (EPFL), Switzerland October 2010 – December 2015

Title: Ph.D. in Computer Science

School and Dates: School of Electrical Engineering, University of Belgrade, Serbia October 2008 – April 2010

Title: Engineer of Electrical Engineering and Computer Science – Master GPA 10.00/10.00
Thesis: Human Computer Interaction Device for Visually Impaired People

School and Dates	School of Electrical Engineering, University of Belgrade, Serbia	October 2003 – October 2008
Title	Engineer of Electrical Engineering Thesis: Tactile Web Browser Simulator	GPA 9.02/10.00

Selected Publications

V. Jovanovic, D. Shabalin, E. Burmako, and M. Odersky, Annotating the Previous Stage: Succinct Type-Driven Staging at Compile Time, Scala'15 (under submission)

V. Jovanovic, A. Shaikhha, S. Stucki, V. Nikolaev, C. Koch, and M. Odersky, Yin-Yang: Concealing the deep embedding of DSLs, GPCE'14

A. Sujeeth, T. Rompf, K. Brown, H. Lee, H. Chafi, V. Popic, M.Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun, Composition and reuse with compiled domain-specific languages, ECOOP'13

T. Rompf, A. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olkotun, and M. Odersky, Optimizing Data Structures in High-Level Programs: New Directions for Extensible Compilers based on Staging, POPL '13

S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky, Jet: An Embedded DSL for High-Performance Big Data Processing, BigData'12

M. Canini, V. Jovanovic, D. Venzano, D. Novakovic, and D. Kostic, Online Testing of Federated and Heterogeneous Distributed Systems, Computer Communication Review, vol. 41, p. 434-435, 2011.

M. Canini, V. Jovanovic, D. Venzano, B. Spasojevic, and O. Crameri, Toward Online Testing of Federated and Heterogeneous Distributed Systems, USENIX'11

Activities

Selected talks	Programming DSLs Made Simple, ScalaDays 2014 Yin-Yang: Transparent Deep Embedding of DSLs, ScalaCamp 2013 High-Performance DSLs Embedded in Scala, GeeCon 2013
Reviewing	Artefact reviewer for OOPSLA'15 Subreviewer for HLPP'14, GPCE'14, and ICFP'14
Demos	Yin-Yang: Concealing the Deep Embedding of DSLs, ECOOP'15
Organizing	Summer School on Domain Specific Programming Languages, Lausanne, July 2015 Scala Workshop, 2013 PL Seminar at EPFL
Teaching	Reactive Programming and Parallelism (2015) Functional Programming Principles in Scala *(2013, 2014, 2015)* Principles of Reactive Programming *(2013, 2015)* Foundations of Software (2012) Operating Systems (2011)

References

Martin Odersky, Professor of Computer Science at EPFL, martin.odersky@epfl.ch

Christoph Koch, Professor of Computer Science at EPFL, christoph.koch@epfl.ch

Tiark Rompf, Professor of Computer Science at Purdue University, tiark@purdue.edu

Leonid Igolink, VP of Engineering, App. Perf. Management at CA Technologies, lim@igolnik.com

Anjan Goswami, Head of Search Science Engineering at Walmart Labs, goswami.anjan@gmail.com