Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming

THÈSE N^{O} 5007 (2011)

PRÉSENTÉE LE 20 MAI 2011

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

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Gilles DUBOCHET

acceptée sur proposition du jury:

Prof. P. Dillenbourg, président du jury
Prof. M. Odersky, directeur de thèse
Dr G. Bracha, rapporteur
Prof. V. Kuncak, rapporteur
Prof. L. Réveillère, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2011

Gilles Dubochet

Embedded Domain-Specific Languages Using Libraries and Dynamic Metaprogramming

Langages dédiés incorporés par l'utilisation de bibliothèques et de la métaprogrammation dynamique

Résumé

La programmation utilisant des langages dédiés à des domaines est certainement destinée à jouer un rôle central en ce qui concerne l'augmentation de la productivité dans l'écriture de logiciels. Cela est dû au fait que les nouvelles abstractions ajoutées aux langages généralistes sont complexes et rarement applicables à des tâches de programmation courantes. Par contre, ces abstractions peuvent servir à encoder des abstractions dédiées. De cette façon, le langage généraliste sert d'hôte dans lequel des fragments dédiés sont incorporés. Les techniques d'ingénierie logicielle habituelles s'appliquent, mais permettent simultanément l'écriture de programmes intégrant divers domaines.

Cette thèse décrit deux techniques de ce type en Scala — un langage orienté-objet et fonctionnel moderne. La première permet d'incorporer des fragments de code ayant une syntaxe dédiée, qui est tissée dans la propre syntaxe du langage hôte de façon à la faire apparaitre comme différente. La seconde technique utilise des types structurels afin d'encoder des propriétés de correction dédiées que des types orientés-objet ne peuvent représenter. Une implantation des types structurels pour Scala est aussi décrite. La caractéristique partagée de ces deux techniques est que leurs propriétés dédiées sont fournies par des bibliothèques définies uniquement en terme d'abstractions généralistes.

Toutefois, ces techniques exemplifient également en quoi l'incorporation de certaines propriétés dépends de données disponibles dans le compilateur, qui sont ensuite perdues dans les langages statiquement typés. Ni les abstractions généralistes, ni les techniques traditionnelles de réflexion dynamique ne donnent accès à ces données ; retenir toutes ces données est également irréalisable. Cette thèse décrit deux techniques de métaprogrammation qui permettent à des bibliothèques d'incorporation de demander l'accès à des données du compilateur sur le code ou les types, selon les besoins. Ces données du compilateur sont alors disponibles comme valeurs dans le programme. Elle décrit par ailleurs une structure de métaprogrammation qui combine des données du compilateur dans le code avec la réflexion. Sa conception est basée sur les miroirs, et permet un partage de code important entre la structure de métaprogrammation et le compilateur.

Mots clés : Techniques et Outils de Conception Logicielle ; Scala ; Programmation Orientée-Objet ; Programmation Applicative (Fonctionnelle) ; Langages Dédiés (Spécifique aux Domaine) ; Bibliothèques Logicielles ; Compilateurs ; Génération de Code ; Métaprogrammation.

Embedded Domain-Specific Languages
Using Libraries and Dynamic Metaprogramming

Abstract

Domain-specific programming is bound to play a central role in improving the productivity of writing software. This is because modern, intricate, general-purpose abstractions are rarely applicable directly to everyday programming. However, such abstractions can be used to encode domain-specific abstractions, so that the general-purpose language serves as a host in which domain-specific fragments are embedded. The benefit of traditional software engineering is maintained, while allowing writing software that integrates multiple domains.

This thesis describes two such techniques in Scala—a modern language with functional and object-oriented features. The first allows embedding fragments of code with domain-specific syntax, which is weaved within the host language's own syntax so that it appears as being different. The second technique uses structural types to encode domain-specific correctness properties that object-oriented types cannot represent. An implementation of structural types in Scala is also described. The common characteristic of both techniques is that domain-specific properties are provided by a library relying only on general-purpose abstractions.

However, the techniques also exemplify how certain embedding tasks rely on data that is available in the compiler but is then lost in statically-typed languages. General-purpose abstractions do not give access to this data, nor do traditional runtime reflection techniques; keeping all data is also impractical. This thesis describes two metaprogramming techniques whereby compiler data on code (code lifting) and types (manifests) can be requested by embedding libraries when needed. Compiler data is then available as literal values in the program. It further describes a metaprogramming framework that combine compiler data available in code with reflection. Its design is mirror-based, and allows a high degree of code sharing between the metaprogramming framework and the compiler.

Keywords: Software Design Tools and Techniques; Scala; Object-Oriented Programming; Applicative (Functional) Programming; Domain-Specific Languages; Software Libraries; Compilers; Code Generation; Metaprogramming.

Contents

Contents 9

Preface 11

I On Domain-Specific Languages 17

1 Domain-Specific Programming: Benefits and Methods 19
1.1 *Related work:* domain-specific programming 22
1.2 Hosting multiple domains 26
1.3 *Related work:* techniques to embed languages 30

2 Using Libraries to Embed Domain-Specific Languages 35
2.1 Growing syntaxes using token cells 37
2.2 Implicit conversions between domains 42
2.3 A grammar of library-embedded languages 47
2.4 The SQL for Scala language 55
2.5 A silver bullet? 57

3 Types That Traverse Domain Boundaries 61
3.1 Structural types in a nominally-typed language 62
3.2 A reflective implementation of structural types 64
3.3 On performance 73
3.4 Type-safe relational algebra 81
3.5 Structural types for domain-specific programming . . . 89

II On Metaprogramming 91

4 Controlling Code and Structure of Programs 93
4.1 *Related work:* staged code 96
4.2 *Related work:* towards a separation of concerns 98

5 Compiler-Supported Code Lifting 101
5.1 Staged compilation 103

5.2 The XQuery for Scala domain-specific language 109
5.3 Discussion and analysis 113

6 Static Types At Runtime **117**
6.1 Type manifests 120
6.2 Revisiting type-safe relational algebra 126
6.3 The manifest manifesto 128

7 A Unified Program Manipulation Framework **131**
7.1 Metaprogramming with mirrors 132
7.2 An implementation to share code 138
7.3 Leveraging unification 145

Conclusion **147**

Bibliography **153**

Curriculum Vitæ **163**

Preface

Immediately after imagining the first computer arose the question on how to communicate with these alien contraptions. To us, people, the tongue that computers understand is very strange. It is impossibly literal—no imprecision or omission is tolerable. Because of that, it ignores hyperbole, images, abstractions: all means that we use to communicate about complex ideas. Of course, if computers were incapable of coping with hard notions, their success would have been limited. But this was not to be the case, and computers have become the beating heart of our information society. The tool that allowed it: programming languages.

Programming languages bridge the gap between people and computers. They allow abstraction, complete small omissions, and find hazardous imprecisions. Because programming languages are so central to using computers, they have been part of their history almost from the start. The intention that underlays most of their development is that of increased abstraction. Concepts such as functions, objects or continuations raise the level of abstraction, giving the developer more leverage to express his ideas. Sixty years of raising the abstraction level have led to modern languages, such as Scala, that allow programmers to express very subtle concepts.

Simultaneously, progress in network and processor technologies have made programming harder today than it ever was. Programs are executed on increasingly elaborate mediums: multi-core processors or the cloud are typical examples. Different parts of programs are on machines of very different capabilities. A web application, for example, is split between the part running on the client's browser and that running on the server. No existing abstractions are useful on all mediums and when code is executed concurrently; efforts to unify different abstractions have been inconclusive to this day.

Furthermore, this happens at a time when adding ever more complex general abstractions in languages seems to yield diminishing returns. Straightforward and intuitive abstractions are already part of most languages. New abstractions, however powerful, tend to be

subtle and require elaborate programming. It is debatable whether they contribute directly to making programs simpler. Must one therefore consider that programming language research is becoming increasingly irrelevant? Is it destined to propose increasingly complex abstractions that are increasingly difficult to use for solving always smaller problems?

No, of course. But abstractions that will help programmers in the future are unlikely to be general-purpose abstractions that can directly be applied to problems, like traditional abstractions have been. For that, they are too complex. Instead, they must serve to enable sophisticated programmers to provide for the needs of others. This fundamentally changes the way in which languages are designed. Now, most programmers are not expected to use new language features, or indeed to understand them. However, they must be able to benefit from them.

To give an example, expert Scala programmers have used *implicit arguments* and *higher-kinded types* to create a collection library that is highly consistent and safe [67]. Users of this library benefit from a depth of type checking that normal abstractions would not permit. Furthermore, the implementation has significantly more code reuse than would have been possible otherwise. Despite that, neither implicit definitions nor higher-kinded types are visible in the user program.

Language research must support the hard work done by specialist programmers using subtle abstractions to deliver straightforward concepts that solve specific problems. A form of programming that is particularly straightforward and specific is embedded domain-specific programming. It is domain-specific because the new abstractions created by expert programmers do not relate to all problems, but are tailored to a specific task in a specific domain. It is embedded because the domain abstractions are hosted within a general-purpose programming languages, with which they integrate. Language abstractions that can be used to create domain-specific languages, and embed them will be the main subject of this thesis.

Domain-specific programming is touted by some [22, 20] as the most promising evolution of modern programming. By giving to programmers tools that are specific to the task they are solving, the gap between the problem and the solution is reduced. Their effort can be concentrated on solving the problem at hand instead of encoding it into generic abstractions. Hard problems can become simple if they are solved using the vocabulary and concepts of their domain. Certain types of problems are already mostly solved using domain-specific

languages: searching text, using regular expressions; or querying relational data, using SQL. However, these domain-specific languages are usually hard-coded or poorly integrated. What abstractions are needed in general-purpose languages to facilitate the development of new domain-specific languages?

Contributions This thesis is an exploration of embedding domain-specific languages in Scala using libraries. It contributes two techniques for writing such libraries. It offers a reasoned discussion on their nature in comparison to traditional libraries, thereby exposing flaws of traditional abstraction for embedded domain-specific programming, and proposes metaprogramming as a solution. Metaprogramming techniques are described; some are new, some are existing techniques improved for domain-specific programming. It contributes a design to unify these techniques into a coherent metaprogramming framework for domain-specific programming. Underlying the whole thesis is the following conjecture, which will be discussed throughout.

> Modern statically-typed object-oriented languages such as Scala have language abstractions—or can support new abstractions—that allow to satisfactorily host domain-specific programming. Domain-specific syntax, semantics, correctness properties and data can be provided without preprocessor or custom compiler. However, more static data must flow to the runtime implementation of domain-specific libraries than what is required by other libraries. Metaprogramming that unifies the compiler and runtime reflection provides the right framework to support this.

The first part of the thesis proposes embedding techniques that rely on existing abstractions in Scala and other modern languages.

- It describes *ZyTyG*, a programming technique that is used to implement a domain-specific language parser as a library in Scala. It also defines *modulo-constrained grammars*, the family of languages that can be parsed using that technique.

- The thesis discusses how *structural types* in the host language can be used to encode domain-specific soundness properties. It describes an implementation of Scala's structural types on the Java virtual machine using reflection. The performance of that implementation technique is analysed and compared with others.

The second part of the thesis presents modifications to the design of Scala that are specific to domain-specific programming.

- The thesis describes *type-directed code lifting*, a form of staged programming inspired by MetaML. Controlling staging using types makes the lifting potentially invisible to the user, which is better suited for embedding domain-specific languages. It presents a design for the implementation of type-directed code lifting in the Scala compiler.

- The thesis introduces *manifests*, a metaprogramming technique inspired by the "scrap your boilerplate" technique, which give runtime access to static type information in languages that do not maintain it at runtime (erased languages). Manifests do not have the overhead of runtime types, yet give enough access to type information for solving an important class of domain-specific embedding problems. The thesis also presents an implementation of manifests in Scala.

- Finally, the thesis describes a mirror-based design for an integrated metaprogramming framework. It combines code lifting, manifests and traditional reflection, bringing together the tools that are needed to solving domain-specific programming tasks. Furthermore, the design unifies related concepts in metaprogramming and the compiler. This allows to dramatically reduce code duplication in the implementation of a metaprogramming-enabled language. It also creates opportunities to better utilise the interaction between the compiler and the runtime to solve domain-specific programming problems.

The work that led to this thesis has been done at EPFL's programming methods laboratory, the home of the Scala language. Much of the discussion will use Scala as example or model. But many of the problems discussed in this thesis are relevant to other languages too, and they may serve as a useful guide for designers of languages that aim to support domain-specific programming.

Part I : On Domain-Specific Languages

The first part of this thesis discusses two examples of using general abstractions to embed domain-specific languages. While the techniques are interesting in themselves—their presentation constitutes the heart of that part—, the topic that underlies it in effect is that of the suitability of general abstractions for embedding. At the

end of this part, we will conclude that embedded domain-specific languages have very particular requirements that are currently not entirely fulfilled by Scala, and generally in statically-typed languages.

Chapter 1 introduces the topic of domain-specific programming and of embedded domain-specific languages. It argues that embedded domain-specific programming is one of the most promising solutions to improve the productivity of writing software. A typology of embeddings is defined, based on their nature and their degree. Current techniques for embedding domain-specific languages are discussed.

Chapter 2 presents ZyTyG, a technique that uses language abstractions in order to imitate a domain-specific syntax that differs from that of Scala. It contains a guide on how to implement the encoding library for all operators of EBNF grammars. The limitations imposed by the technique are described through a new class of regular grammars, and the corresponding state automata. The technique is exemplified by its use to embed ISO SQL, whose implementation characteristics are briefly described.

The theme of type-safe domain-specific programming is discussed in Chapter 3, which postulates that certain characteristics of structural types make them particularly suitable for typing domain-specific code. Scala's structural types are shortly described, and their implementation in the Scala compiler is presented, and compared to another implementation technique. A system to guarantee soundness for a relational algebra domain-specific language using structural types is described. It results in the observation that, while type soundness can be provided, the domain-specific language cannot be implemented to generate actual values.

Part II : On Metaprogramming

The second part builds on the weaknesses described in existing abstractions that can be used to embed domain-specific languages. It describes metaprogramming techniques that give domain-specific embedding libraries access to information about the program that they require. Two specific techniques are described, one pertaining to accessing code, the other types, and concludes by attempting to unify them.

Chapter 4 reviews the previously-described techniques to understand the fundamental nature of the problem afflicting them. The proposed solution is a form of metaprogramming whereby more static information is made available through runtime reflection. Techniques

providing related forms of metaprogramming through staging are discussed, as well as state-of-the-art design techniques for reflection.

The first component of this work's metaprogramming proposal, which concerns code, is laid out in Chapter 5. The concept of lifted code is described, as well as its implementation in Scala. A method to use types to control lifting is outlined. The usage of the method is described though an example of an embedded XQuery library, where queries are written in native code that is transformed by a domain-specific library at runtime.

Chapter 6 introduces type manifests, the second component of the metaprogramming proposal. Various aspects of their use and of their integration with the language are explained. An overview of their implementation is provided. The unfinished example of Chapter 3 is completed with the help of manifests, demonstrating their contribution to domain-specific programming. The last section of this chapter grapples with the question of whether a language remains statically-typed in the presence of such metaprogramming.

The final chapter of the second part (Chapter 7) observes that the previously described metaprogramming techniques become truly powerful when working together and with runtime reflection. It proposes to extend the unifying idea of mirror-based reflection to encompass also the compiler, which code lifting and manifests strongly relate to. The question of maintaining identity in such systems that span multiple metaprogramming providers is discussed. An implementation of mirror-based metaprogramming in Scala is proposed, first concentrating on its ability to foster code reuse, which is later extended to help integrating multiple frameworks into one.

The conclusion chapter of this dissertation reviews how supporting embedded domain-specific programming changes the way in which one considers the purpose of a language. Certain operations that in normal programming are deemed dangerous or useless become crucial to successful embeddings. A series of recommendations for language design in light of this understanding are presented.

Part I

On Domain-Specific Languages

Chapter 1

Domain-Specific Programming: Benefits and Methods

A problem is hard in a given language if solving it requires the programmer to display an unusually high level of creativity, or if it requires an unusually high number of trials. As an example, writing high performance code is usually a hard problem. Even though general design principles exist—use of caching, parallelism, known high performance algorithms—the design space that must be explored is very large. Finding the right implementation for a problem, requires broad knowledge, judgement and often a fair amount of cunning. Furthermore, because it is difficult on modern hardware to have a good mental model of performance characteristics, writing a high performance program usually requires many iterations of benchmarking and adaptation. In other words, a priori knowledge cannot be used to make correct design decisions most of the time.

On the other hand, easy problems are such that their solution is like their description. A striking example is user interface design in WYSIWYG development, where both the solution and its description are identical. We will discuss this example below. Writing a language parser using a parser combinator library is easy because, from the grammar—its description—, the implementation is trivial. Easy problems may still require large programs to be solved, for example when writing content-driven web applications. However, their uniform and predictable nature makes them easy despite their size.

Whether a problem is easy or hard obviously depends on its nature. Some problems such as simple arithmetic on relatively small numbers are intrinsically easy on a computer. For most problems, however, their difficulty depends upon the programming model in which they are solved. In other words, it is the quality of the

abstractions provided by the programming language or environment that make a problem be easy or hard.

The quality of abstractions in modern languages is much improved over those of their ancestors. Let us examine Scala as an example of a modern language with varied and powerful abstractions. It unifies the paradigms of object-oriented and functional programming. Higher order functions and other abstractions considerably reduce its syntactic overhead and allow the program to focus on its core characteristics. Scala's static type system provides safety to programs and allows to define rich interfaces for better modularity. Martin Odersky and other proponents of Scala claim that programs written in Scala can be as little as half the length of equivalent Java programs. I have shown [32] that shorter programs lead to easier and faster code comprehension in an experimental setup, even when the complexity of the algorithms they represent is equivalent.

However, the long term trend of languages becoming more expressive is counterbalanced by the fact that programming becomes intrinsically harder. In particular, programming environments are more complex today than they ever were before. Single applications rely on multiple platforms with different capabilities and whose interconnection is often complex. This is particularly noticeable in large web applications that span the browser, server infrastructure—itself oftentimes distributed—multiple databases, et cetera. However, even traditional desktop applications no longer are developed on a unified platform. GPU and cloud computing are becoming mainstream techniques, sapping the old programming paradigms. Instead of a single, homogeneous system, developers have to take into account the inherent complexity of uneven capabilities and interconnection that heterogeneous applications requires.

In other words, improvements to programming languages are counterbalanced by the increasingly complex nature of modern programs. Which of these two factors is more important is an interesting question that is very difficult to answer convincingly and beyond the scope of this dissertation. However, it is clear that simply enhancing the way programs are written without fundamentally changing what is written, allows incremental improvements at best. Only when incremental improvements in languages collude to make hard problems easy can breakthroughs happen.

What abstractions can make a hard problem easy? Of course, answers to this question include a component of taste, and are therefore necessarily personal to some degree. However, we will discuss below

a popular idea that may be a general solution to this problem. This idea is domain-specific programming. The assumption is that, once the domain of a problem has clearly been identified, and a domain-specific language exists to program it, that problem becomes easy. The end of this section explores this assumption.

To start with an example, we will discuss the task of programming user interfaces. The interactive graphical user interface, introduction in the 1970s, changed the way programs are written. With traditional batch programs and programs with semi-interactive text based interfaces, the substance of any code is in its business logic. On the other had, laying out interface widgets and defining their interactive behaviour became an important part of the complexity of new graphical programs. Using general-purpose abstractions for this task does not work well. The linear nature of code is fundamentally in conflict with:

- their two-dimensional layout;
- values that are inherently relative—this widget is left-aligned with that other—as opposed to absolute and numeric;
- their interactive behaviour.

Because of that, programming environments were extended with programming models specific to user interfaces. As early as the mid-1970, a quintessentially domain-specific solution started to gain in popularity: WYSIWYG user interface editors. There, the problem of programming a user interface is reduced to the task of drawing it. Apple's HyperCard, released in 1987, is an example of early graphical editors for graphical user interfaces. Other later examples of such tools are Microsoft VisualStudio or Borland Delphi, whose impact on programming practices of the decades of 1990 and 2000 was profound.

Figure 1.1 shows the help screen for adding a new button to a user interface in HyperCard. Notice how most of the described interaction has to do with domain-specific notions, such as moving the button, changing its size, or linking to another card. Only the notion of "editing its script" relates to a concept outside of the domain. This is symptomatic of a domain-specific solution to a problem, which relies on abstractions and on a vocabulary that are part of the domain. No unrelated abstractions encumber the solution to the problem, as is the case when using general-purpose mechanisms. In fact, the description of the solution—a display of the desired user interface—is the solution itself. This is domain-specific programming at its purest. Anyone, a child, can write a user interface using HyperCard. The hard problem becomes easy when domain-specific abstractions are used.

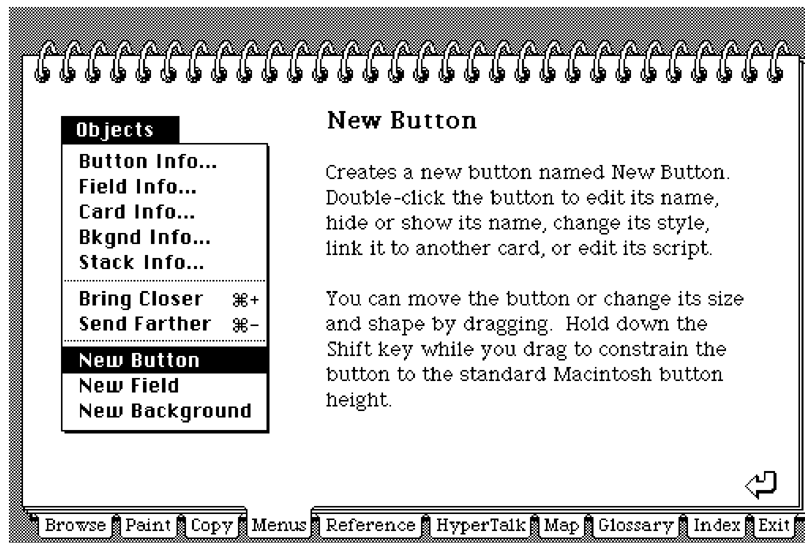


Figure 1.1: HyperCard help screen describing how to add a button to a user interface.

However, while interesting to demonstrate the benefit of domain-specific programming, HyperCard and other graphical interface editors are not immune to critique. To start, the way in which these tools bind behaviour and layout is necessarily constrained. This prevents the definition of any interface with non-standard user interaction. More fundamentally, however, these tools invert the focus of program design found in traditional general-purpose programming environments. A problem that is merely an element of the larger program in traditional approaches, becomes the primary structure around which the program is designed. The program is built around its user interface, even when that structure does not suit the problem well. Not surprisingly, applications developed using graphical interface editors are not renowned for their refined design.

1.1 *Related work:* domain-specific programming

A usual definition of domain-specific programming is that it allows to solve only one family of problems. Its techniques are usually unsuited for programming other domains. In reality, however, domain-specific programming must be seen as a matter of degrees rather than as a strict segregation. To try to better understand its nature, this section will outline the state of the art of domain-specific programming techniques. However, because the field is so broad, it does not aim

to be an exhaustive review of related literature. Instead, it aims to highlight general trends with a selection of relevant works.

Domain-specific programming is a technique that is applied to a variety of programming methods and tools. The following are some examples of the forms it can take:

- graphical languages;
- textual languages;
- libraries;
- development environments;
- analysis tools and framework.

A very thorough review of domain-specific programming techniques can be found in [65], with the exception of graphical languages. In [56], another attempt at classification focuses on a set of domain-specific features and their availability in various tools. As of Wile's article [96], it contains an analysis of the pitfalls specific to varied domain-specific techniques.

Libraries By the definition above, most libraries are domain-specific because they use the vocabulary of the domain they model. In fact, any end-user program is domain-specific. This gives another hint towards the pervasive presence of domain-specific thinking in the very act of programming. However, it is not of great interest to our discussion, because it makes of domain-specific programming an intractable problem. Only when domain-specific libraries are considered in a very restricted sense, are they interesting targets for research, such as in the examples below. The notion of "active libraries" [93, 28] was coined to describe libraries that take part in their own compilation. A recent article by Guyer and Lin [39] demonstrates how domain-specific semantic properties of libraries can be utilised by the compiler for optimisation.

Domain-specific languages Throughout this thesis, our minds will focus on one central domain-specific programming technique: domain-specific *languages*. Indeed, this dissertation's approach for domain-specific programming is to build on top of existing programming systems, and most such systems are based on textual languages. By concentrating on domain-specific languages, we scope the problem in such a way that makes it possible to solve.

Once again, we must observe that, to grasp domain-specific languages requires nuance. There is no definitive or standard technique to create domain-specific languages. In fact, certain designs of libraries go far enough in changing code that uses them so that it behaves as a domain-specific language. A programming technique qualifies as a language if it controls some or all of its syntax, semantics, and correctness properties. This gives it a strong enough identity to be much more interesting than mere libraries.

Domain-specific languages are an old idea. In 1966, Peter Landin discusses a "family of unimplemented languages" that can be instantiated for a given domain [55]. While this article does not mention domain-specific languages, it postulates that "the next 700 programming languages" would be designed with particular aptitudes towards particular areas through their physical appearance and logical structure. Landin's approach to language families has however not seen wide acceptance, although modernised forms of the idea regularly reappear. For a recent view of these efforts in generating new languages, see Czarnecki's overview [27].

Predicting the importance of domain-specific languages, and their impact on programming habits, is a recurring theme. Recently, Taha, in a short but far-sighted pamphlet [87], argues that domain-specific languages will bring programming into everyday lives because their clear relation to their domain makes them easy to use for everyone. He exemplifies this idea by describing a household formalised as a series of needs in terms of grocery lists, recipes, health, etc. Without going that far, domain-specific language had been recognised as an important programming technique for a long time. For example, Bentley noted how programmers employ what he called "little" languages [7] as a technique to solve specific problems, which he exemplifies in a language to process surveys. The idea that domain-specific languages would become a central programming paradigm is implicit in the popular term of "fourth generation" languages—following second and third generation, general-purpose languages. Confirming the argument in the introduction of this chapter, "fourth generation" languages have been reported as significantly more productive than equivalent general-purpose languages, for example in Klepper and Bock's 1995 study [49]. It must however be noted that I am not aware of very recent studies comparing the relative benefits of domain-specific languages and modern general-purpose languages. In the field of software engineering, the idea of "software factories" [26]—a development processes modelled on the efficient 1970s Japanese factories—led to analysing the role that domain-specific languages can play in

such standardised designs. Amongst other, Steve Cook emphasises the agility that domain-specific languages bring to software factories [20, 21]. In Microsoft Visual Studio's conception of software factories, domain-specific languages play an important role [22, 94]. Researchers considering specific domains of application have also proposed domain-specific languages to overcome increasing complexity and to reduce defects, for example in communication services. Consel and Réveillère propose a domain-specific language paradigm for defining communication services [19]. There domain-specific language directly represent protocols such as IMAP, guaranteeing conformance with the protocol while supporting application-specific variations. Burgy, Réveillère et al. propose a lower-level approach [13] named Zebu, which uses a domain-specific language inspired by RFC specifications to define new protocols. There, domain-specific knowledge can be used to verify certain properties of protocols, and to generate relatively efficient parsers.

With a broad agreement on the use of domain-specific languages, researchers started considering their specific nature when compared to general-purpose languages. The multiplication of domain-specific languages, and their inherently unique nature, led to an important literature of experience reports. For example, one can consult the proceedings of the yearly workshop on domain-specific modeling [77, 76], or those of the workshop on domain-specific program development [57]. The general trend in these reports is to favour hybrid graphical and textual environments. Vandeursen et al.'s annotated bibliography [92] concentrates on textual languages, and contains a a list of example domain-specific languages in the fields of financial products, multimedia, telecommunication, etc.

Graphical languages The previous chapter contained an example of a graphical programming environment for a graphical domain. However, these systems are not limited to domains that are themselves graphical. The following two examples relate to domains that are not . In 1975, Zloof described a graphical language to query relational database "by example" [97]. In a completely different domain, Spectra [89] is a graphical tool to program the elements that compose a software-defined radio. Graphical domain-specific environments are popular, and many are developed every year. However, this thesis will leave these systems aside.

1.2 Hosting multiple domains

In the previous chapter, we discussed complete domain-specific languages. Such languages provide by themselves all that is needed to solve a domain-specific task. However, this thesis will not consider complete languages. Indeed, a typical software system is composed of various subproblems. Many of these subproblems are domain-specific, but not all are in the same domain. Composing modules, defining a graphical user interface or querying data are domains found in almost every modern software system. Other domain-specific problems are less common but may represent a considerable part of a program's value: calculate physical forces on objects, perform statistical analysis of data, implement a communication protocol or communicate with hardware devices.

To take a concrete example, a spreadsheet application will have a graphical user interface, may store and retrieve data using SQL databases and needs to parse cell expressions to evaluate them. Each of these three tasks becomes easy if a corresponding domain-specific language can be used to solve it. However, because multiple domains exist in a single application, multiple domain-specific languages must coexist in the code. This rules out traditional monolithic domain-specific languages which replace the general purpose programming model by their own. To contain and integrate the different domains composing the application, a shared programming infrastructure is required. It must provide for all needs of the various domains, like a *substrate* holding and nourishing plants.

A common solution is to consider each domain as a module in the program, and the problem of bringing them together as one of module composition. The substrate itself is then provided as a domain-specific language for module composition. For example, Java EE web applications are typically composed of a WAR archive. It contains compiled Java and XML files defining various elements of the web application, each using specialised libraries. The component elements are brought together in the `web.xml` configuration file that assigns each element to a predetermined role. This file is written using XML according to a domain-specific schema—which is arguably not the most attractive example of a domain-specific language, but that does not influence the substance of this example. Its format allows roles such as generating a page at a given location, as well as more specialised roles such as controlling security or accessing databases. Each component is coded in a single domain—output a web page or control access—without relation to the rest of the program. The

application server configured by the `web.xml` file creates a substrate that allows the components to communicate.

However, this approach suffers from its inflexibility. We recall that the power of domain-specific programming comes from reducing the general programming task to that of configuring predefined domains. As an example, a `web.xml` file configures access control by providing one or many filters that check whether a given URL can be accessed. But because this domain forces access control to be defined in terms of URL, the infrastructure cannot be used to control access to subelements of a page. More generally, by making the substrate a domain-specific language forces the application to be in that domain. This considerably limits the virtues of domain-specific programming as laid out at the beginning of this chapter. Indeed, the argument goes that almost any program contains subproblems that benefit from domain-specific programming. Because programs are so diverse, the argument would become considerably weaker if reduced to only those categories of programs for which a domain-specific substrate language exist.

For this reason, instead of composing domain components with a domain-specific language, a general purpose language should be used. This allows arbitrary components to be integrated and allows them to be extended or connected using arbitrary logic. Furthermore, it does not require every component to be written using domain-specific code, but allows structuring the program using the general purpose language and select domain-specific languages only where they provide the most benefit. However, it raises the question of what makes a general purpose programming language fit for this purpose. The means through which general purpose languages can host domain-specific components will occupy us throughout most of this dissertation. To start this discussion, we will discuss the nature of domain-specific embeddings, and the form taken by embedded domain-specific languages in a general-purpose host language.

The nature of embeddings When speaking of any programming language, it is necessary to separate the following aspects of the program:

1. its *syntax* is the words that can be used and their structure;

2. its *semantics* is the way in which domain-specific fragments are evaluated;

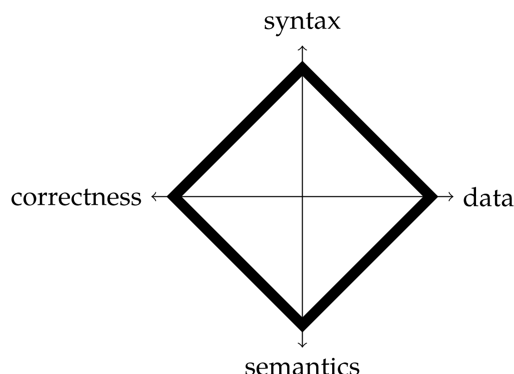
3. its *data* is the form and behaviour of values in domain-specific fragments;

4. its *correctness* are properties, usually defined in terms of types or grammatical structure, that allow to guarantee certain properties of fragments a priori;

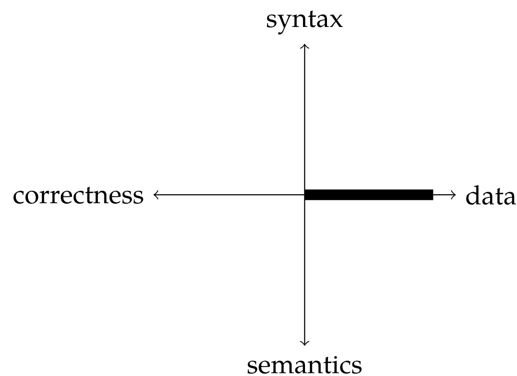
Both domain-specific and general purpose languages are defined in terms of all four of these aspects. However, a domain-specific language embedded in a general purpose host need not define all aspects for itself. It is usually preferable when aspects from the general purpose host can be preserved. This simplifies the process of writing embedded domain-specific languages, and concentrates changes where they matter most. Depending on the domain that is being modelled, different aspects are concerned. This explains why there is no unique solution for embedded domain-specific languages.

Furthermore, the degree of modification required for an aspect can vary. When considering semantics, one domain may require a single element of the evaluation to be changed, another may require the semantics to be changed completely. For example, a domain-specific language to support parallel programming may provide a simple parallel **`for`** loop, which only requires the semantics of this operation to change. On the other hand, a more complete language that supports arbitrary parallelism may require changing the semantics of almost all operations so that they support transactions, shared memory et cetera.

For the sake of our discussion, the nature and the degree of change between the host and the embedded domain-specific language can be represented using the following graph. A domain-specific language that does not share any element of the host would be represented as follows.



On the other hand, a domain-specific "language" that is composed of a library only corresponds to the following situation.



This corresponds to the usual domain-specific capabilities of most existing general purpose languages. On the other hand, the former case is the "golden quadrilateral" of full support for domain-specific programming. In the following two chapters, which compose the first part of this dissertation, I will present two techniques that aim to maximise the surface of domain-specific embedding supported by Scala. Neither offers the full quadrilateral of the optimal solution but both propose triangular surfaces missing only one aspect of domain-specific embeddings. In Chapter 2, correctness is missing or limited while Chapter 3 specifically concentrates on the question of soundness. The limitations observed in these chapters will call for extensions to the host language specifically to allow full embeddings, which will be the subject of the second part of the dissertation.

An embedding that redefines all elements of the host language is obviously very flexible. This allows the domain-specific language to be tailored exactly to the needs of the domain. However, there is also a strong benefit in reducing the embedding surface. That is, it ought to modify the smallest possible number of aspects of the host language and in the least possible manner. The reasons are twofold. First: one reason to embed domains as opposed to using standalone languages is in order to reuse the host language's infrastructure. Second, and more fundamentally: by making the embedded domain completely separate from the remaining of the program, interaction between the two parts becomes less straightforward. For example, if the embedded domain system has its own type system that does not rely on that of the host, defining soundness of domain-specific code will require additional machinery to interface it with the host. On the other hand, if soundness of the domain can be defined in terms of host language types, the relation will be automatic. This is why, in practice, domain-specific embedding techniques must not maximise the potential scope of their embedding, but also allow flexibility in defining how much of it is used for a given embedding.

No matter how shallow the modification called by a domain, the point where domain-specific and general-purpose code meet forms a boundary. An embedding technique must provide for its definition. The table below lists the typical nature of issues that arise for the four aspects of embeddings.

Aspect	Typical issues
syntax	Detecting fragments of code that are in domain-specific syntax.
semantics	No major issues if domain-specific operations are disjoint from those of the host. Otherwise, similar to syntax.
data	If data from the host program is meaningful in domain-specific fragments, it should be accessible as domain-specific data. Data representation must automatically be converted between domains.
correctness	Similar issues as in syntax and data embeddings. Furthermore, may require defining one type discipline in terms of the other to obtain a notion of overall type safety.

Of course, as in any exercise of language design, the tasks described above ought to be solvable in such a way as to require as little effort as possible. For programmers using an embedded domain-specific language, the solution should be transparent and not require any knowledge of the embedding technique. This means that the boundary conflicts should all but disappear as if domain and general-purpose code exists in a completely integrated system. For those embedding new domains, it should be lightweight to create specific languages. For both, the process should be as safe as possible.

1.3 *Related work:* techniques to embed languages

The previous section laid out the reasons for preferring embedded over self-contained domain-specific languages, which raises the question of what embedding technique to employ for what purpose. In highly dynamic languages such as Scheme [83] or Ruby [24, 25], embedding techniques are relatively well understood. However, in languages with a less flexible syntax, and particularly statically-typed languages, the question of how to best embed domain-specific languages remains. Broadly, embedding techniques can be separated into the following families:

1. custom preprocessors and compiler front-ends;

2. macro systems and embedding frameworks;
3. staged metaprogramming;
4. library-based integration.

Custom solutions To this day, the usual approach to embedding in statically-typed languages remains the use of custom preprocessors. The original source contains interleaved host language and domain-specific code. The latter has different syntax, semantics and correctness properties, which the host language compiler cannot understand. Therefore, the preprocessor modifies the source before it is seen by the compiler of the host language. By replacing domain-specific fragments with host language expressions that encapsulate the domain-specific behaviour, it generates a new source that the host language compiler can utilise.

As a first step, the preprocessor must detect the boundaries of fragments written in the domain-specific language. Because the preprocessor is executed before the compiler, it does not have access to the structure of the program. Any understanding of the structure that may be required to discover the boundaries must be calculated by its own logic. This is why, in practice, most preprocessors require that domain-specific fragments be enclosed by unique character sequences that render the task of finding boundaries trivial.

This approach makes the preprocessor blind to the environment in which the domain-specific fragment exists, which does not allow to tightly bind the fragment and the rest of the program. To overcome this blindness, preprocessors usually rely on two methods. A first solution is to optimistically assume that the environment provides whatever values or functions are needed to correctly rewrite the domain-specific fragment. The transformed host language code implicitly contains the assumptions made about the environment in the form of references to values for example. When compiled by the host-language compiler—or when it is executed—the assumptions are either validated or an error is produced. However, because the assumptions are tested against the transformed program, errors cannot be reported in terms of the domain-specific code. In a second solution, the preprocessor will restrict the interaction between the domain-specific fragment and the rest of the program. Only values of a restricted set of types that map directly to the domain can be referenced in the fragment.

Because of the limited understanding of the program's structure by the preprocessor, programs are awkward to write and not very

safe. Nevertheless, in many cases, the benefits of domain-specific programming outweigh those costs. Oracle's Pro*c/c++ [70] is a rather old-fashioned but popular embedding of SQL in C and C++ code, exemplifying the strength and weaknesses of preprocessor-based embeddings.

To overcome some of the limitations of preprocessors, it is possible to preprocess domain-specific fragments as part of the front-end of the host language's compiler. By doing that, it is possible to detect domain-specific fragments simultaneously to parsing the host language's own grammar. This gives more flexibility in detecting the beginning and end of fragments. Scala's XML syntax [34] is an example of a domain-specific language embedding in a compiler's front-end. The domain-specific parser may also have more access to compiler-generated data about types and names. However, parsing source code must usually be completed before the compiler can calculate program properties. A compiler extension that does domain-specific transformation relying on such properties would have to juggle with mutually dependent compiler and domain-specific data. This makes it a very subtle program.

Specialised embedding tools Macro systems provide tools to define preprocessor-like transformation on source code. The C preprocessor, for example, can be used with domain-specific programming in mind, although it remains a very crude tool. Other textual transformations tools, such as `sed` or `awk`, are also often used as preprocessors. However, such systems do not utilise any knowledge about the structure of the program to verify macro transformations, or to provide useful abstractions.

So called "hygienic macros" were originally proposed by Kohlbecker et al. to guarantee that macros do not violate the language's identifier binding discipline [50, 17]. Although originally for LISP, similar hygienic propositions appeared on macro systems for statically-typed languages. The evolution from unsafe textual macros to systems acting on the syntax tree or on other high-level language elements is laid out in an article by Brabrand and Schwartzbach [10]. The JTS tool [6] preprocesses Java code according to external transformation procedures described in the JAK language. Other systems such as MS [95] or JSE [5] use macro instructions interleaved with source code, like the C preprocessor, but acting at the level of syntax. A system with a more general purpose, but widely used to embed domain-specific languages is the C++ template macro infrastructure [3]. Finally, MetaBorg [12] is a domain-specific language embedding framework,

which uses rules to define syntactic rewrites and specialised tools for program transformation.

Languages with staged metaprogramming stand out amongst hygienic rewrite tools. They offer a level of power and of safety unequalled by other systems. I will however postpone the discussion on that technique to Part II of this dissertation, where it will feature prominently.

Embedding using libraries Instead, let us go back to domain-specific languages using a simpler embedding infrastructure. In fact, the last family of embedding methods we want to discuss does not use any tool beyond existing general-purpose language abstractions. Although used in many programs for a long time, Hudak first made explicit the properties of this embedding technique [42], which he called "modular monadic interpretation". No specialised tool rewrites a program's syntax to transform domain-specific fragments. Instead, it is data structures that convey the look and feel of syntax. The domain-specific embedding library interprets the data structure as if it were the syntax itself. Others have extended Hudak's interpretation technique to generate c++ code instead [45]—effectively compiling the embedded fragments—or to make the interpretation more flexible and support optimisations [40].

However, the syntax of domain-specific languages embedded using Hudak's technique remains very close to that of the host. This is in many cases an advantage, as Leijen and Meijer discuss in a proposed embedding for a database querying language [60]. However, to make it a broadly usable embedding technique, more work is needed on the possibility to modify the host language's syntax. This will be the subject of the next chapter.

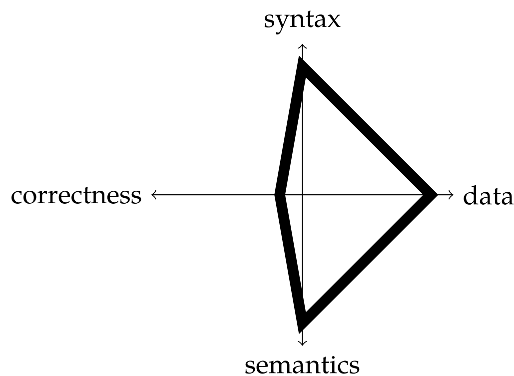
Chapter 2

Using Libraries to Embed Domain-Specific Languages

Scala offers abstractions that act on syntax, and others that allow to design complex libraries. Amongst the former are techniques like infix operators defined as methods, or variable-length argument lists. Amongst the latter are implicit conversions, which allow to convert objects from one type to another automatically and based on need. In the absence of these abstractions, as is the case in Java, syntactic embeddings are very limited [36]. But by leveraging these tools, this chapter discusses a method to embed a domain-specific language with a specific syntax, using only libraries.

I originally introduced the technique [31] under the name of "ZyTyG" (Zygotic Type-Directed Growth). Its key idea is that embedded domain-specific fragments will remain valid expressions in the host language. From a programmer's perspective, their look-and-feel will be domain-specific. However, they will be executed as Scala expressions, thereby reconstructing a representation of the domain-specific fragment as intended by the programmer.

In the classification of embedded domain-specific languages described in the previous chapter, ZyTyG covers the following surface.



The technique's primary goal is to provide domain-specific syntax to act on domain-specific data. As was mentioned above, the normal evaluation semantics of the host language lead to creating a representation of embedded fragments, which must then be given their own domain-specific semantics. In simple cases, these semantics can be represented in terms of operations on data structures of the host language. This is typically the best solution if the domain-specific language is used to describe such structures. For example, consider a language of time expressions like "`1 hour 13 minutes`" that directly represent instances of a `Time` class. Alternatively, the library can provide a domain-specific interpreter that allows arbitrary semantics for the embedded language. However, this entails a runtime penalty that may not suit certain use cases.

While ZyTyG offers a relatively high flexibility in terms of embedded syntax and semantics, it currently lacks a convincing ability to test correctness. To some limited degree, the conformance of domain-specific fragments to correct syntax can statically be guaranteed through the host's type system. However, ZyTyG is quite inept at guaranteeing the soundness of the *behaviour* of domain-specific fragments. Indeed, as we will see, host language types are utilised to provide the domain-specific syntax. Whilst these can be extended by parameters that represent domain-specific typing properties, soundness can only be checked if the syntactic form of the embedded language matches the typing discipline of the domain-specific language.

Another obstacle with ZyTyG is that the languages that it can represent are limited, as their grammar must be compatible with that of the host language. In reality, this problem is less overwhelming than intuition would have us believe. First, the family of embeddable grammars can formally be defined. This facilitates reasoning about their design. Second, many languages are naturally inside of—or very close to—this family, which seems to correspond to a natural form of

domain-specific languages. For example, we will see in §2.4 that SQL is a member of this family, with very small exceptions.

Despite its restrictions, we want to consider ZyTyG because it is a domain-specific technique that does not rely on specialised language abstractions. It covers an aspect of embedded domain-specific languages—syntax—that is usually provided using preprocessors. Instead, ZyTyG aims to utilise already existing abstraction in order to cover as much of the embedding surface of domain-specific languages. We will see that the resulting embeddings are more flexible than those obtained by most preprocessor or even macro systems, and similar in nature to those found in dynamically-typed languages. Furthermore, ZyTyG, or variations thereof, are now commonly used; the attraction of domain-specific programming is so strong that many Scala developers overlook its restrictions. By describing ZyTyG, and by further formalising its properties, it it possible to better grasp the reality of embedded domain-specific programming today.

2.1 Growing syntaxes using token cells

The first half of this section describes how to design a library that embeds a given domain-specific language from a syntactical point of view. The second half discusses how domain-specific expressions should be constructed so that the code of the expression can be retrieved. Strategies that make the boundary between the domain and the host more permeable are discussed in the next section. This technique is implemented using Scala as the host language. To my knowledge, no other statically-typed language currently supports all necessary features to implement ZyTyG. It is of course part of my argument that languages wishing to support domain-specific programming ought to consider these features.

Growing a syntax The idea underlying this technique is to define the grammar of a domain-specific language in terms of a sequence of host language identifiers for objects and methods defined in the embedding library. Each object or method identifier corresponds to one closed token in the embedded language. A token is said to be closed if it has a unique form—for example keywords. This differs from open tokens, which can take multiple forms—for example numbers—and that will be discussed in the next section. The objects defined in the library and representing keywords are called "token cells" and contain one or more "token methods" following a structure

that is discussed below. A set of token cells with their methods defines the grammar of an embedded language.

To start, we shall consider how to embed languages whose only valid sentence is fixed. The language defined by grammar G_1 is such a language.

G_1 := `a b c`

This language can be embedded in Scala by using the following library.

```
object a {
  def b(next: c.type) = next
}
object c {}
```

In this example, the closed tokens `a` and `c` are defined as cells, while `b` is defined as a method. Cell `a` also serves to mark the beginning of the domain-specific fragment. Cell `c` is the end of the fragment. Note that, at this point, nothing prevents partial fragments that do not reach `c`.

If the syntax for method calls in the host language were like that of Java, the library above would obviously not parse G_1. For it to work requires the host language to support calling single-parameter methods as infix operators. In other words, it requires that the expression `a b c` be equivalent to `a.b(c)`. This is the case in Scala, originally with the idea of supporting specific domains such as mathematical expressions. Here, we can see how this feature is more broadly useful to domain-specific programming because it allows to break out of the method-calling paradigm of the host language and subvert it to embed another language.

The type `c.type` of the `next` parameter of method `b` is a Scala singleton type. Only two values are of that type: the object named `c` and `null`. This type guarantees that a unique cell instance is used to parse token `c`. The fact that Scala singleton types also allows `null` is not desired in this case because the parser can thereby exit in an uncontrolled manner. Nevertheless, it provides static guarantees on the identity of cell instances that will become more relevant later, once cells are extended with parsing state.

Longer sentences can easily be parsed by adding a method to object `c` on the model of that added to `a`.

```
object a {
  def b(next: c.type) = next
}
```

```
object c {
  def d(next: e.type) = next
}
object e {}
```

By using the same system as many times as needed, it is possible to write grammars for any arbitrary sequence of closed tokens, assuming the following two criteria are met:

1. that each token is a legal value or method identifier in the host language;
2. that the number of tokens in the sequence is odd.

Of course, fixed sentence languages are only of limited use; any meaningful grammar will at least contain choice and repetition structures. The two languages defined by EBNF grammars G_2 and G_3 contain slightly different forms of choice structures.

G_2 := a (b c | d e)
G_3 := a b (c | d)

Both can easily be encoded as a library. Grammar G_2 simply requires that token cell `a` defines a separate method for both of its successor tokens: `b` and `d`.

```
object a {
  def b(next: c.type) = next
  def d(next: e.type) = next
}
```

Grammar G_3 requires a host language mechanism so that multiple object types can follow `b`. Various solutions are possible, for example using subtyping. However, in terms of its real-life behaviour, the most satisfactory implementation uses overloading and yields the following implementation.

```
object a {
  def b(next: c.type) = next
  def b(next: d.type) = next
}
object c
object d
```

Grammar G_4 and G_5 are equivalent but demonstrate two methods of obtaining a repetition in an EBNF grammar.

G_4 := a { b a }
G_5 := a | a b G_5

Implementing such a repetition in the language library is done by using the host language's own recursion mechanism.

```
object a {
  def b(next: a.type) = next
}
```

With these techniques, the three basic structures of EBNF grammars—sequence, choice and repetition—can be encoded in a library for the host language.

You may notice that, because the parsing is defined in terms of host language types, the compiler will be able to detect various syntax errors in the embedded language. However, these errors are reported in terms of the structure of the embedding library, which may be confusing. The following table lists a number of typical errors when using the parser of grammar G_1.

Expression	Error
`a b c`	No error (correct expression).
`a b x`	Not found: value `x`.
`a x c`	Value `x` is not a member of object `a`.
`a b a`	Type mismatch; found: `a.type`; required: `c.type`.

Furthermore, if a token cell is used in various locations in the program, the type of that cell must accept all derivations from any of the locations it is used. In that, the quality of type checking will decrease if the same token is used in multiple places in the grammar. For example, a library that encodes a grammar containing token `c` at two separate positions will also accept fragments that are missing all tokens required between the two `c` tokens.

Building a representation The technique described above allows fragments of domain-specific code to be embedded in the host program and accepted by the host compiler, without any preprocessing. To continue the embedding requires to create a useful representation of the fragment.

When the expression that encodes a domain-specific fragment is executed, the resulting value is whatever token cell was used last. This defines the point reached in the grammar at the end of the parsing. For some grammars, such as G_1, G_2 or G_3, knowing the end point is sufficient to reconstruct the entire domain-specific expression. In most cases, however, more information is needed. In particular, the number of iterations must be stored in case of repetitions, and the path

taken by choices in which branches do not remain strictly separate must be recorded. The first case can be seen in grammars G_4 and G_5, whist the second is demonstrated in G_6 below—which also contains a repetition.

G_6 := a { (b | c) a }

A step in the parser happens when a method is called on the previous token cell, passing it the next token cell and thereby consuming two tokens in the grammar. For example, in grammar G_1, if the parser is in the state of token cell `a`, a step is when method `b` is called on `a` with parameter `c`, thereby consuming tokens `b` and `c` and leaving the parser in the state of token cell `c`. If that step corresponds to a repetition or to a choice, the information that needs to be recorded is temporarily available in the body of the method. Storing it in a state field of the resulting token cell will make it available at the next step, where it can be forwarded in the same manner. At the end of parsing, the token cell returned to the host will define the end point in the grammar by its type, and the choices and repetitions made during the parsing will be available in its state field.

The following example encode the language defined by grammar G_6 in such a way that the host program can reconstruct the parsed domain-specific fragment.

```
class A {
  var state: List[String] = Nil
  def b(next: A) = {
    next.state = "b" :: state
    next
  }
  def c(next: A) = {
    next.state = "c" :: state
    next
  }
}
def a = new A
```

In practice, as we will discuss in §2.3, it is usually more practical to return in the state the entire list of parsed tokens, including those that could be reconstructed from the final parsing point.

The embedding technique describe above is attractive because it allows domain-specific programming in languages completely different from the host using only libraries. For a developer, there are no additional constraints in terms of tooling or workflow when compared to any other library. Furthermore, because the host language guarantees

safe separation between libraries, it is easy to use different embedded domain-specific languages, as long as they are strictly disjoint.

The technique, however, puts strong constraints on the family of grammars that can be used to define domain-specific languages. The supported grammars are formally categorised in §2.3; we will see that, while limited, they have the advantage of being easy to work with. Another constraint comes from the fact that the host language restricts value and method identifiers. For example, and for obvious reasons, a method cannot be called "`(`", which prevents embedding parenthesised languages. However, by nesting grammar fragments, it is possible to use host language parenthesised expressions within domains-specific code. We will see in §2.4 that, with a little ingenuity and very few compromises, it is conceivable to embed complex, real-life languages.

2.2 Implicit conversions between domains

This section is concerned with the boundary between domain-specific expressions and host programs. We will discuss three questions pertaining to domain boundaries:

1. how to utilise token cells and token states from domain-specific expressions to execute domain-specific behaviours;

2. how to use values or expressions from the host program by adding open tokens to domain-specific expressions;

3. how to use the logic of the host program to parameterise the structure of domain-specific expressions or to reuse fragments thereof.

Question 1: Executing domain-specific behaviours Depending on the domain, the domain-specific expressions embedded in the host program will have to be calculated, transformed, interpreted, shipped to a third-party tool, etc. However, the details of that process are very much dependent on the domain being considered and beyond the scope of our discussion. For the time being, we can simply assume that the domain-specific library provides a function that calculates the result of a domain-specific expression given a token cell and the stored information about the parsing path.

The only relevant property of that function is whether it returns a value for the host program, or not. A value is returned, for example, in an SQL query expression. In this case, the function will take the SQL expression, transform it into a string, send it to the database engine,

and return a set of relations to the host language. On the other hand, a value is not returned in an SQL delete expression. In that case, the function will again send the string of the query to the database engine. But the function has no meaningful result—assuming errors are reported using exceptions. The purpose of the domain-specific expression comes from its side-effect.

If a value is returned, that value can serve as the glue between the domain-specific expression and the host program. This technique requires the host language to have abstractions that automatically convert values based on their types, which Scala does using implicit conversions (see §6.26 in [66]). Implicit conversions are specially marked, single-parameter methods that the compiler will automatically add to code if that code cannot be typed and if adding the conversion makes it typeable. For example, consider the following definitions.

```
implicit def x2y(x: X): Y = new Y
def f(y: Y) = ...
```

Without implicit conversions, the expression `f(new X)` will not compile and report a type mismatch between `X` and `Y`. With conversions, however, the expression will be compiled by automatically modifying it to `f(x2y(new X))`.

In the embedding library, implicit conversions are defined from all token cells that correspond to a final state of the parser, to the resulting host language value. These conversions calculate the host language values resulting from the domain-specific expressions. Because implicit conversions in Scala are simply defined as methods, they can implement the function that calculates the domain-specific expression. In the example below, the implicit method `evaluate6` evaluates a fully-parsed expression of grammar G_6 to a host language `Int` value.

```
implicit def evaluate6(last: A): Int = {
  ... // evaluate domain-specific expression
}
val dsResult: Int = a b a c a
```

Because the domain-specific expression `a b a c a` is valid in G_6—its last state is of type `A`—, the compiler automatically applies it to `evaluate6`. The domain-specific expression is therefore computed to a host language value.

The advantages of this solution are multiple. It is very simple to use. The relevant implicit conversions are imported with the rest of the embedding library. Because the user does not see the conversions,

which are automatically inserted by the compiler, the domain-specific expressions feel perfectly embedded. The solution is also relatively safe. Because the designer of the library has been careful to only define implicit conversion from final parse states to host language values, any domain-specific expression that is incomplete is reported as erroneous by the compiler. Similarly, if the logic of the host program expects a value that the domain-specific expression cannot provide, there will be no available conversion and an error is reported. However, these errors suffer from a similar problem as those reported against the syntax of the domain-specific expression by the embedding library: they relate to the embedding library, not to the domain-specific code. Errors such as "type mismatch; found: `a.type`; required: `Int`" will be generated. An error like "domain-specific expression `'a b a'` does not yield a value of type `Int`" would have been preferable.

If no value is returned by the domain-specific expression, which only exists for the purpose of its side effects, the strategy above does not work. One solution is to require the expression to be enclosed in a computation method expecting a final parser cell, or that such a method be explicitly called on the cell. For example, let us extend the final state of grammar G_6 with a computation method.

```
class A {
  ...
  def evaluate: Int = {
    ... // evaluate domain-specific expression
  }
}
def a = new A
(a b a c a).evaluate
```

Method `evaluate` serves the same purpose as the `evaluate6` implicit conversion above. However, incorrect behaviours can silently be accepted by the compiler if the user forgets to use the computation method. In that case, neither will the compiler check that the expression is in a final state, nor will it be evaluated at runtime.

Question 2: Accessing host values Accessing values from the host program from within the domain-specific expression is similar to the previous problem. Values from one domain must be transformed into another with the help of translation functions. It will therefore be solved in a similar fashion.

In the previous section, a domain-specific expression is described as a sequence of identifiers defined in the embedding library. Identifiers are either token cells (objects), or methods. Methods are fixed,

but for token cells, the logic of the embedding does not necessarily require that the cell be identified directly. Instead, it suffices that the host language value that is located at that point evaluates to a cell. In particular, it is possible to use a host program value as long as there exists an implicit conversion from that value to the correct cell. In other words, by defining an implicit conversion from a host language type T to a token cell, that token cell can be used to accept an open token covering all values of type T.

For example, consider grammar G_7 that is not only composed of tokens but also comprises one value.

G_7 := `a b` *number* `d e`

This value can either be written in the domain-specific expression as a constant or as a reference to a number within the host program. Examples of valid expressions are "`a b 4 d e`" or, assuming that "`four`" is defined in the host program, "`a b four d e`". An embedding library for this language can be written as follows.

```
object a {
  def b(next: c.type) = next
}
object c {
  def d(next: e.type) = next
}
object e {}
implicit def number2c(n: Int): c.type = c
```

By defining an implicit conversion from all host language integers to token cell `c`, the latter becomes an open token for numbers in the grammar.

This logic even allows arbitrary host-language expressions to be used to calculate the value, for example in "`a b (4*four) d e`". Cells that are implicitly constructed from values in the host program and that correspond to open tokens are called "value cells". The obvious drawback of this approach is that value cells are only usable in the grammar where token cells are allowed. We will discuss possible solutions in the next section.

Question 3: Parametric expressions Finally, we must remember that, with the technique we have been discussing, embedded domain-specific expression are also valid host language expressions. More precisely, for every two additional tokens corresponding to a parsing step, a new host-language expression is defined that returns a new token cell object. This cell corresponds to the state of a partially parsed

domain-specific expression. It is quite possible to directly manipulate token cell values in the host program to parameterise the structure of the domain-specific expression.

For example, assume we want to write a domain-specific expression in grammar G_4 where the number of repetitions depends on the state of the host program.

$G_4 \quad := \quad$ `a { b a }`

The implementation of G_4 as an embedding library with state information can be done as follows.

```
object a {
  var repetitions = 0
  def b (next: a.type) = {
    next.repetitions = repetitions + 1
    next
  }
}
```

A valid expression in G_4 for an arbitrary number of repetitions "`numRep`" can be generated using a fold operation, such that at every step a new token cell is generated and passed to the next. In this example, every token cell is final in the grammar, but this is not required: it is quite possible to have the host program work on non-final states of the domain-specific expression.

```
(1 to numRep).foldLeft(a)((prev, n) => prev b a)
```

Another use of the same technique is to build commonly-used fragments of domain-specific expressions and complete only the remaining portion when needed. For example, assuming a domain-specific language for SQL, it is possible to store SQL fragments that select from known tables in the database, but without defining any condition. At different places in the program where specific selections are needed, the original fragment can simply be completed with the necessary condition. This powerful ability to work with domain-specific languages goes far beyond what is conceivable in traditional preprocessor embedding techniques.

We have seen how implicit conversions can be used to entirely define the boundary between domain-specific expression and the host program. When compared to alternative explicit solutions, implicit conversions are well suited for this task because they balance the conflicting requirements of well-defined yet transparent domain limits. On one hand, the boundary is constrained and safe because it is entirely established in terms of the implicit conversions provided

by the embedding library. On the other hand, integration can be very tight because implicit conversions work at the level of types and are therefore impervious to the manner in which the shared values are obtained. In fact, we can here see a first example where statically-available types are beneficial over dynamic ones for domain-specific programming.

2.3 A grammar of library-embedded languages

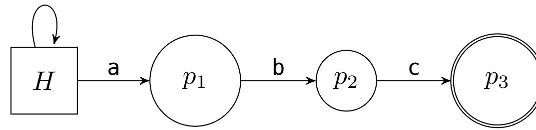
We have seen in the previous two sections what constraints are put on the grammar of domain-specific languages by the embedding technique. As we will see, some of these constraints may be relaxed by further stratagems employing additional language features. However, reasoning about embeddable languages is already thorny without added complexity. I try to demonstrate in this chapter that library-based embeddings can be used for complex domain-specific languages. These are languages such as ISO SQL, whose ZyTyG-based embedding is discussed in the next section. To reason about such a complex grammar requires a formalism encompassing the constraints of ZyTyG. For this purpose, this section introducing a new class of grammars called "modulo-constrained regular grammars". Further relaxations to ZyTyG's constraints will be discussed within its framework.

The informal limitations observed during the description of ZyTyG can be summarised as follows:

- tokens are parsed using one of two separate techniques: cells (objects) and methods;
- both cells and methods can parse closed tokens—tokens that correspond to fixed keywords;
- only cells can parse open tokens—tokens that may be any of a class of values;
- for the moment, let us assume that parsing can be described in terms of a deterministic finite state automaton, meaning that the technique can at least parse regular languages.

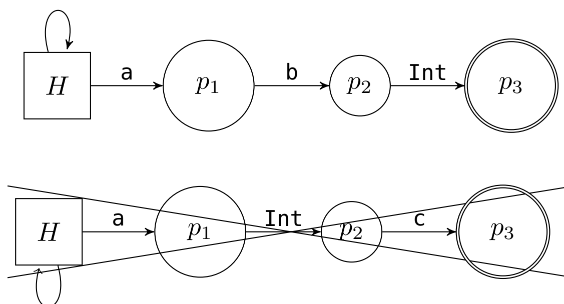
To start, let us consider the last statement in more detail. We had previously used a notion of "step" to describe the evaluation of a parser, similar to a step in an automaton; the main difference being that a step consumes two tokens. In contrast, below is a description of

a finite state automaton that accepts the same language as grammar G_1 (`a b c`).



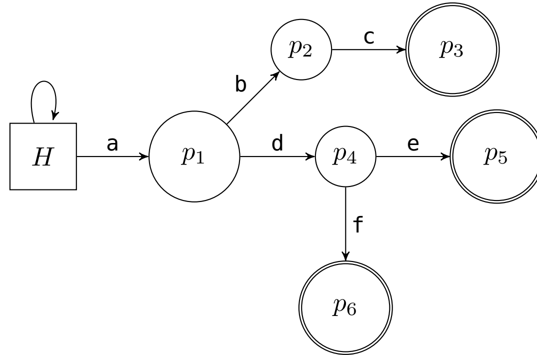
The initial state H is the state in which the parser is available—when the embedding library is in scope—but where no domain-specific expression is being parsed. As long as the starting marker `a` is not consumed, the parser remains in that state. Upon consumption of token `a`, the parser moves to state p_1, which corresponds to the instantiation of token cell `a`. Consumption of token `b` happens through the selection of method `b` on token cell `a` and leads to state p_2. Note that this state is drawn as a smaller circle to indicate that it does not correspond to a token cell. Finally, token `c` is consumed, which happens by the argument being passed to the previous method. The state p_3 reached at this point corresponds again to a token cell, which explains the larger circle. It is also a final state for which an implicit conversion back to a host value exists, which is represented by a double circle. The steps of the parsers are recognisable as double transitions between large states.

If the language to be parsed contains open tokens, such as arbitrary numbers of other values, they must be implemented as cells. If the open token defines a transition from a small to a large state, it corresponds to a token cell in the implementation. On the other hand, if it defines a transition from a large to a small state, it cannot be implemented as an embedding library. Consider for example the following two automata, where `Int` is the set of all Scala integer values. The first yields a language that can be parsed using ZyTyG, the second does not because it has an open transition between a large and a small state—however, we will later see that extensions to ZyTyG may actually allow it.



The choice and repetition structures that can be used in an embedding library correspond again to specific structures in an automaton. Consider grammar G_8, which corresponds to the automaton below, and demonstrates choice.

G_8 := `a ( b c | d (e | f) )`



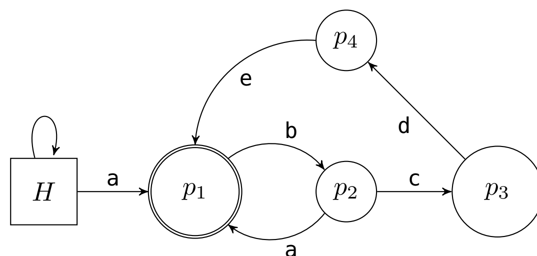
From state p_1, which corresponds to a token cell, a deterministic choice is made based on reading token `b` or `d`. A choice is also made from state p_4, which corresponds to a method. In the case of a token cell state, each outbound arrow is implemented as a method of a different name. In the case of a method state, each outbound arrow is implemented as one overloaded alternative for the same method. In other words, the embedding library can provide choice from both kinds of states.

```
object a {
  def b (next: c.type) = next
  def d (next: e.type) = next
  def d (next: f.type) = next
}
```

While not demonstrated in the example, it is also trivial to provide choice on the first token.

Now consider grammar GI, which corresponds to the automaton below, and demonstrates repetition.

GI := `a { b a | b c d e }`



This example contains two loops, one on two tokens (`b a`), the other involving four (`b c d e`). Until now, the implementation of the embedding library imposed a constraint on the parseable language only in terms of the location of open tokens. In the presence of repetitions appear new constraints on the structure of the underlying automaton. To implement the GI grammar, the following library can be used.

```
trait p1 {
  def b (next: c.type) = next
  def b (next: a.type) = next
}
object a extends p1
object e extends p1
object c {
  def d (next: e.type) = next
}
```

We notice that, for a single state p_1, multiple token cells must be defined (`a` and `e`). On the other hand, the implementations can be shared through inheritance from `p1` because the method tokens available from that state are shared between the cells. This is because in an embedding library, token cells serve two purposes: to define a token in the grammar and to hold the set of following tokens. In the example, the set of following tokens is shared, but the tokens themselves are distinct.

However, only repetitions on sequences of even length are allowed. It is easy to see that a repetition on a single token cannot be implemented because that token would have to be simultaneously implemented as a token cell and as a method. Of course, this doesn't prevent a repetition on the same token twice, as in "`{ a a }`", but that is not identical to a single token repetition. In terms of finite-state automata, this property can be described by saying that transitions may only go from a big state to a small one. In particular, this prevents transitions to the same state, which are how single token repetition would be represented. Furthermore, the very first token of the grammar, which marks the beginning of the domain-specific expression, cannot be part of any repetition.

Modulo-constrained grammars To formalise the constraints imposed by embedding libraries, I will introduce a new class of grammars called "modulo-constrained grammars". A modulo-constrained grammar defines a language that can be accepted by an embedding library as defined above, or by a deterministic finite state automaton with big and small states. In a modulo-constrained grammar, each

token is annotated with one of a c or m marker. The c marker indicates that reading this token leads to a big state, that is, it is implemented as a cell. Conversely, an m token leads to a small state and that it is implemented as a method. For example, grammar GI would be written as follows.

$$GI \quad := \quad \mathtt{a}_c \; \{ \; \mathtt{b}_m \; \mathtt{a}_c \mid \mathtt{b}_m \; \mathtt{c}_c \; \mathtt{d}_m \; \mathtt{e}_c \; \}$$

For a grammar to be modulo-constrained, all accepted phrases must yield a sequence of tokens such that only c-marked tokens are at odd positions and only m-marked tokens at even positions. Furthermore, all final tokens must be c-marked. To maintain this property between production rules, non-terminal symbols are also marked. The following is a rewrite of grammar GI using intermediate non-terminals.

$$\begin{aligned} GI_c^c &\quad := \quad \mathtt{a}_c \; \{ \; HI_c^m \; \} \\ HI_c^m &\quad := \quad JI_c^m \mid KI_c^m \\ JI_c^m &\quad := \quad \mathtt{b}_m \; \mathtt{a}_c \\ KI_c^m &\quad := \quad \mathtt{b}_m \; \mathtt{c}_c \; \mathtt{d}_m \; \mathtt{e}_c \end{aligned}$$

All non terminals are annotated with two markers. The marker in the superscript defines the state that is reached after reading one token into the non terminal. The marker in the subscript expresses the state that is reached at then end of reading the non terminal. Note that it is also justifiable to speak of superscript and subscript markers in tokens. But because they are always the same, the superscript marker can be omitted for tokens.

For reasons of simplicity, we will consider the modulo constraining properties assuming an EBNF grammar such that:

- all repetitions contain a single, non-terminal symbol;
- all branches of all choices contain a single, non-terminal symbol.

However, the properties are straightforward to generalise for other grammars. For a grammar to be modulo-constrained requires the following six properties to hold true.

1. The superscript marker of all non terminals must be the same as the superscript marker of all symbols in its first set.
2. The subscript marker of all non terminals must be the same as the subscript marker of all symbols in its final set.
3. If G is the start symbol of the grammar, it must be annotated as G_c^c.
4. If H is a non terminal, and there exists a repetition in which it is used (such as in "{ H }") it must be annotated with H_c^m.

5. If J and K are two non terminals, and there exists a choice in which both are used (such as in "$J \mid K$") they must both have the same superscript markers and the same subscript markers.

6. All open tokens must be marked with c.

For a language to be parsed by an embedding library requires two properties.

1. A regular, modulo-constrained grammar that generates the language must exist. This property is fundamental to the embedding strategy.

2. All closed tokens must be identifiers that are available in the host language and, for all open tokens, there must exist a type in the host language that contains all values accepted by the token. How restrictive these properties are is very dependent on the host language.

In practice, of the two restrictions on tokens, it is the availability of identifiers that is the most limiting.

Context free grammars We will discuss in the next section whether the limitations laid out above still allow parsing meaningful domain-specific languages. However, before we do that, there remains one issue. Because the formalism is first described in terms of finite-state automata, it only insures that modulo-constrained grammars are *regular*. In reality, ZyTyG can at least parse context-free grammars by using the state stored in cells to encode the stack of a pushdown automaton. In other words, there are no additional constraints on the EBNF form of modulo-constrained grammars than those described above. In practice, however, it is cumbersome to manually manage the state needed to parse context-free grammars. By following the example of what is done in traditional parsers, we can come up with a better solution to parsing embedded domain-specific languages.

Clearly, it is not necessary that the language be entirely defined in terms of a ZyTyG parser. Most language parsers in compilers are implemented as a two phase process starting with a fast scanning phase and followed by a more capable parsing phase on the result of the scanning. The scanning phase usually transforms streams of characters into tokens and values using a regular grammar. The parsing phase is described in terms of a deterministic context free grammar, which is typically implemented using a LL- or LR-based parser. A similar approach can be used for embedding libraries. By treating the embedding library as a scanner, which aim is only to

generate a stream of token, it is possible to use a friendlier parser afterwards, such as Scala's parser combinator library.

In this approach, the state field of token cells in the embedding library store a sequence of tokens. At each step, two tokens are added to the state: the method token and the token of the cell that was just entered.

$$GI \quad := \quad \mathsf{a}_c \; \{ \; \mathsf{b}_m \; \mathsf{a}_c \mid \mathsf{b}_m \; \mathsf{c}_c \; \mathsf{d}_m \; \mathsf{e}_c \; \}$$

For example, an embedding library scanner for grammar GI above may be implemented as follows, here using Scala symbols as tokens. If the scanned language contains open tokens, a richer data structure for tokens would have to be used.

```
trait Tokens {
  def tokens: List[Symbol]
}
class Scanner(token: Symbol) extends Tokens {
  var tokens: List[Symbol] = List(token)
  def stepInFrom(pToks: List[Symbol],
                 mTok: Symbol): this.type = {
    tokens = token :: mTok :: pToks
    this
  }
}
trait p1 extends Tokens {
  def b (next: a.type) = next.stepInFrom(tokens, 'b)
  def b (next: c.type) = next.stepInFrom(tokens, 'b)
}
object a extends Scanner('a) with p1
object e extends Scanner('e) with p1
object c extends Scanner('c) {
  def d (next: e.type) = next.stepInFrom(tokens, 'd)
}
```

Trait `Tokens` and class `Scanner` are generic implementations that can be reused. The actual embedding library is almost identical to the prototypical implementation. Only minor adjustments are required to store tokens.

The complete token string for the domain-specific expression will be available in the implicit conversion that transforms the final token cell into a host language value. By using an arbitrarily complex parser, it is possible to parse higher-level properties on the domain-specific grammar. However, in this model, all of the parsing only happens when the host program is executed. This implies that the ability of

the compiler to statically detect syntax errors is limited to those that pertain to the scanner.

In parenthesis A particular problem encountered when embedding domain-specific libraries is that of parentheses. Because of the limitations of the parser, it is not possible to directly parse them. However, the meaning of parenthesis hardly ever changes: their role as a marker of priority or relation is quite universal. Because of that, it is likely that the semantics of parentheses in the host language can be reused for the same purpose in the embedded language.

For example, assume we want to parse the language defined by grammar GJ, which contains an expression in parentheses. This can be done by nesting a sub-language inside a host language parenthesised expression, which is then parsed by its own parser. The trick works if the whole parenthesised expression is located at a c-marked position in the modulo-constrained grammar.

$GJ \quad := \quad$ `a b '(' c d e ')' f g`

```
object a {
  def b (next: x.type) = next
}
object c {
  def d (next: e.type): x.type = x
}
object e
object x {
  def f (next: g.type) = next
}
object g
```

Upon completion of the nested expression parser, its result is forced to a nested expression cell `x`. This cell represents the whole parenthesised expression in the outer grammar. This implementation allows obtaining better type checking characteristics, for example by correctly rejecting the sentence "`a b e f g`". However, it requires that the end of the nested expression can be detected by the host type system. In a language where the final cell of the nested expression also is a non-final one, an implicit conversion from the nested expression to the nested expression cell must be used. This is for example the case if the nested expression concludes in a repetition.

Another available trick to use host language parenthesis is by calculating a token cell not by a value identifier in the embedding library, but by a method call. This allows to use at a c-marked position

a closed token further followed by a parenthesised expression. An example of such a grammar is GK, implemented below.

GK := a '(' b c d ')' e f

```
def a (sub: d.type): x.type = x
object b {
  def c (next: d.type): d.type = next
}
object d
object x {
  def e (next: f.type) = next
}
object f
```

Finally, both tricks used to support parenthesised expression rely on the parenthesised form of method argument lists. If the host language supports multiple arguments, it becomes quite easy to parse multiple, comma-separated subexpressions in the parenthesis. Furthermore, in Scala, method argument lists may also be enclosed in braces. This allows the same tricks as described above for expressions nested in braces. However, the parser cannot differentiate between expressions nested in braces or parenthesis.

2.4 The sql for Scala language

It remains to be seen whether modulo-constrained grammars allow to encode useful languages. To do so, we will consider a class of languages often found in domain-specific programming.

Pseudo-human languages define the structure of their sentences not by operators and recursively nested subexpressions, but by using relatively fixed word sequences. While less flexible than traditional programming languages, they are quite popular for domain-specific programming because they are considered more user friendly. I will show in this section that the methods described in the three previous sections are powerful enough to encode sql, a complex example of pseudo-human languages, almost exactly according to its specification. The embedding library described in this chapter has been implemented by Cédric Lavanchy under my supervision [58].

ScalaDBC is a wrapper around the JDBC database library and part of the Scala standard library. Unlike JDBC, ScalaDBC represents queries as native data structures—as opposed to strings. Sql's syntactic elements are represented by classes whose members are the sub-elements, effectively building a syntax tree for queries. However,

writing this directly is verbose and unintuitive. For example, the query “SELECT * FROM persons” would be written as follows:

```
statement.Select {
  val setQuantifier = None
  val selectList = Nil
  val fromClause = List(statement.Relation {
    val tableName = "persons"
    val tableRename = None
  })
  val whereClause = None
  val groupByClause = None
  val havingClause = None
}
```

Instead, a ZyTyG library allows writing queries with SQL syntax. The previous query is then written as “select * from ’persons”. In fact, the domain-specific language supports all of SQL’s query and database modification language with only very minor modifications required to the syntax to make it modulo-constrained. This allows writing queries such as the following.

```
select ('age, 'name, 'salary)
from ('persons naturalJoin 'employees)
where (
  'gender == "M" and
  ('city == "lausanne" or 'city == "geneva"))
orderBy 'age
```

The few differences imposed by ZyTyG on the grammar have to do with reserved keywords in the host language. For example, selecting a field of a table in SQL is done using the dot as operator, as in “persons.name”. Scala does not allow to overload the meaning of the dot; ScalaDBC therefore requires to use another operator and write “persons**name”. Another example is that reference to arbitrary tables or fields are not tokens in the domain-specific language, but instead Scala values of type Symbol.

The SQL ZyTyG library is implemented using a variation of the scanner plus parser approach described in the previous section. The scanner is defined in two separate parts. One part of about 150 lines of code provides all entry points to SQL—starting tokens and implicit conversions. The other, of about 350 lines, contain the remainder of the scanner. Unlike the design described earlier in this chapter, the SQL scanner does not create one class for every object token. Instead, it has a single ExpressionBeyond class that represents all object states and

contains all methods. This simplifies the design of the scanner and, by reducing object instantiation, improves its performance. However, it reduces the quality of static errors related to the domain-specific language. The parser utilises the token stream generated by the scanner and implements the full ISO SQL specification. It uses Scala's parser combinator library and is composed of about 140 non-terminals in 750 lines of code. The implementation is remarkably compact If one considers that it encodes a sizeable fraction of the second part of the ISO SQL specification [44], a document of over 1200 pages.

2.5 A silver bullet?

Throughout this chapter, we have seen how the syntax of a domain-specific language can be embedded using only libraries. The technique is not trivial, and suffers from certain restrictions. However, to assess its merits, one must compare it to traditional embedding techniques.

- Traditional techniques have the advantage of unlimited flexibility in the design of the embedded language. There are no restrictions to the language family like those imposed by modulo-constrained grammars. Furthermore, because the embedding happens statically, it does not cause any runtime overhead—the quality of the transformation to host language expressions remains a concern, however.

  On the other hand, the use of preprocessors, dedicated macro systems or custom compilers renders the programming environment more complex. Moreover, soundness properties are rarely defined on the domain-specific code. This is because domain-specific soundness depends upon the remainder of the program, which macro systems or preprocessors have very little access to.

- ZyTyg simplifies the programming environment: existing IDEs, debuggers and processes work for domain-specific code; there is no need for additional tools in the compilation chain.

  But the key advantage of using libraries to embed syntax is the level of integration allowed between domain-specific code and the host program. Domain specific code can easily and safely access values from the host program. In fact, general purpose expressions can be inserted into the domain-specific expressions to calculate values. A small restriction to mixing domain-specific code is that Scala cannot generate multiple implicit conversions that are required to convert data from one domain to

another. This requires explicitly defining conversions between these domains. Despite this caveat, implicit conversions give a flexibility in terms of handling domain-specific interpretation that is extremely costly to replicate in traditional methods.

Because domain-specific expressions are also host-language values, such expressions can easily be constructed programmatically. For example, different fragments of domain-specific code stored as values can be joined into a useful expression by the host program depending on its state at that point. This ability is in effect a sort of code generation ability that improves reuse. An approach like ZyTyG also allows for a more gradual extension of the embedding. The embedding language may for example first be used only for a subset of the domain-specific problems, the remainder defined as domain-specific data structure using the host language's syntax.

A superior embedding strategy for domain-specific languages would merge the benefits of ZyTyG and traditional solutions, while shedding their flaws. As was discussed in §1.3, modern takes on traditional techniques explore macro systems tightly linked to the compiler, which better integrate domain-specific fragments with the remainder of the program. However, these systems are inherently complex because they solve the problem by adding new abstractions or tools. ZyTyG starts from the principle that it is preferable to solve a problem without an additional layer of tooling. Furthermore, the flexibility afforded by library-based embedding in terms of integration and reuse seems very difficult to provide using traditional techniques.

Can one start from a library-based embedding technique and make its syntax more flexible, give it better correctness properties, and make it faster? Incremental improvements to ZyTyG are certainly imaginable. For example, Rompf experimented with an extension that models the stack of a pushdown automaton, thereby providing additional static verification for the domain-specific syntax [74].

The next chapter moves away from ZyTyG to discuss a different embedding strategy, which may end up answering some of ZyTyG's restrictions in terms of correctness. However, we will also conclude that, fundamentally, domain-specific embeddings require access to static data that isn't available during runtime of statically-typed languages. On the other hand, domain-specific embedding techniques that rely on the compiler or a static macro system solve only one side of the problem, missing the dynamic nature that gives library-based solutions an edge in terms of integration. To tackle all facets of the

problem will requires linking dynamic and static characteristics of the program in a way that isn't afforded by usual language abstractions.

Chapter 3

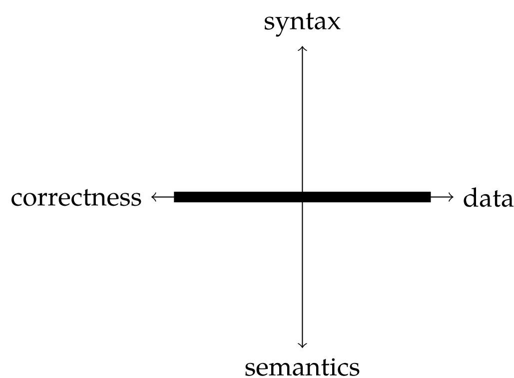
Types That Traverse Domain Boundaries

We have seen in the previous chapter how to use the host language's syntax and semantics to encode a domain-specific language. Notably, however, this technique hijacks the type system which is no longer available for its original purpose. It becomes all but useless for checking soundness properties of domain-specific expressions, and their interaction with the host program. Even a dynamically-typed version of ZyTyG would require information about the environment that is only accessible through metaprogramming, as we will see in the second part of this thesis. Instead of pursuing this route, this chapter steps back from syntax to consider using domain-specific types for domain-specific programming. This is an easy problem if the host language's types can represent the domain directly. Object oriented types are well suited in that regard. But certain domains are inherently incompatible with that model, even with enriched object types like those of Scala. Domain-specific languages that access somewhat unstructured data are oftentimes a poor fit.

Take for example a domain-specific language for accessing data in a relational algebra [18], like SQL. Common implementations, such as JDBC, are poorly embedded and provide no guarantees on types at all. Object-relational mapping systems deliver a degree of safety by fitting the types of the relational algebra to those of objects. However, because object types are more restricted, an object-oriented view is forced upon the relational data. As an example, Hibernate [47] represents a relation as a class, and its rows as objects. But a relational algebra, using one of its "join" operators, allows to create a new relation by merging two. Because classes cannot be joined in the same manner, object-relational mappings do not support this part of the algebra, forcing a certain view upon the data to the detriment of

any other. This example demonstrates how subtle domain-specific properties—here the ability to arbitrarily choose associations between relations—are lost when mapping a domain-specific type system to that of the host language.

In order not to suffer from such restrictions requires the host language's type system to offer types powerful enough to encode domain-specific properties. The usual approach is to provide stronger control on types through additional kinds and operators. Instead, I postulate that many domains in fact require types that are weaker, implying less control. A sloppier type system may provide the added flexibility needed for embedding domain-specific expressions. In this chapter, we will discuss *structural types* as an alternative in domain-specific programs to traditional nominal types. As an embedding technique, it obviously concentrates on domain-specific correctness properties—specifically soundness—, which also implies a relation to the shape of data.



3.1 Structural types in a nominally-typed language

Traditional object-oriented type systems require to explicitly name the type of every object by giving it a class. Similarly, subtyping relations between types are made explicit by using the **`extends`** keyword when declaring a class. Because they name all "is a" relations explicitly, such type systems are called "nominal". Conversely, structural types, also known as "duck typing", allow a type to be subtype of another if their structures — that is, their members — allow it. At the simplest, a type is allowed to be a subtype of another if it exposes at least the same members.

Structural and nominal types are not mutually exclusive. Amongst other, the calculus νObj [68], which was presented in 2003, includes both nominal and structural subtyping in a unified way. Malayeri and Aldrich later proved that the unification of nominal and structural

types is sound using their *Unity* calculus [64]. Scala concretises Odersky's ideas presented in νObj, although early versions of the language lacked full structural types; only refinement types were available. There, the structural aspects taken into account by types is limited to the argument and result types of existing members of a class. However, as of version 2.7 of Scala, full structural types are available.

To illustrate Scala's structural types, let us consider an example. Some objects have a `close` method that must be called after use. Without structural types, writing a generally applicable `autoclose` method would require all closable objects to implement an interface containing the `close` method. Structural types solve the problem differently.

```
type Close = Any { def close: Unit }
def autoclose(t: Close)(run: Close => Unit): Unit = {
  try { run(t) }
  finally { t.close }
}
```

To be compatible with the "`Close`" structural type, an object must be compatible with `Any`—all objects in Scala are—and it must implement the `close` method. It is the structure of their members—name, type of the arguments and of the result—that define whether an object is of a given type. The `autoclose` method is implemented using the `Close` type. Its arguments are: first, the object to be automatically closed, such as a socket or file, and second, some code that uses the object and must be run before the object can be closed.

```
autoclose(new FileInputStream("test.txt"))( file =>
  var byte: Int = file.read
  while(byte > -1) {
    print(byte.toChar)
    byte = file.read
  }
)
```

The closable object that is passed to `autoclose` is an instance of `FileInputStream`. It conforms to the structural type because it contains a `close` method. `FileInputStream`, which is part of Java's standard library, does not implement any particular interface related to type `Close`. The second parameter for `autoclose` is a block of code that will read the stream. After executing it, `autoclose` will `close` the file. The method call `t.close` must be executed from the body

of `autoclose`. At that location of the program, the fact that `close` is defined in the object `t` is known because of the structural type.

3.2 A reflective implementation of structural types

Structural types are are a general-purpose construct. This section discusses their implementation, the next one considers its performance. Structural types are abstractions that go beyond what can directly be mapped onto the Java virtual machine. This and the previous chapter hinted towards the fact that abstractions for domain-specific programming may often display a similarly flexible nature. In that sense, a discussion on the implementation of structural types is likely representative of the sort of challenges that arise when implementing other abstractions for domain-specific programming.

A Scala method call is normally directly compiled to an equivalent JVM method call instruction. This instruction requires that the JVM knows beforehand that the receiver of the call can respond to it—for reasons of performance. Because structural types are not part of the JVM, it does not know that `t` can respond to `close`. This shows that calls to methods which are statically defined as members of a structural type (structural method calls) cannot simply be compiled to a JVM method call. Instead, they require a different compilation technique that bypasses the constraints of the JVM. Two families of compilation techniques can be used to this end: hybrid generative-reflective techniques and pure reflective techniques.

Hybrid generative techniques create Java interfaces to stand in for structural types on the JVM. The complexity of such techniques lies in that all classes that are to be used as structural types anywhere in the program must implement the right interfaces. When this is done at compile time, it prevents separate compilation. When that is done at runtime, it requires to adapt dynamically objects to interfaces, which is expensive and complex.

Gil and Maman's Whiteoak [38] is an extension to Java with structural types that are compiled using a generative technique. Whiteoak does not modify classes to implement interfaces at compile time; instead, it uses ASM, a bytecode generation framework, to create wrapper classes at runtime. These wrappers implement the interfaces corresponding to structural types and forward all method calls. Whenever a structurally defined method is to be called on an object, a wrapper for the structural type is generated and the method is called on the wrapper instead of the object. The wrapper then delegates the call to the original object. A different wrapper class is needed for

every structural type—because its interface changes—and for every type of delegation object—because the code of the forwarder methods is tailored for one specific delegation. Whiteoak uses a global cache strategy to reduce the number of wrappers that must be generated.

Pure reflective techniques replace JVM method call instructions with Java reflective calls. Reflective calls do not require a priori knowledge of the type of the receiver: they can be used to bypass the restrictions of JVM method calls. The complexity of such techniques lies in that reflective calls are much slower than regular interface calls. Optimisations are required for this technique to be competitive. The first part of §3.3 analyses the performance of Scala's compilation technique of structural types. Its second part compares the performances of Scala's reflective compilation technique to that of Whiteoak's generative technique.

Compiling structural types The compilation technique of structural types in Scala that is discussed below is based on a reflective technique. To overcome the cost of reflection on the virtual machine, it uses varied caching strategies. The overall structure of the technique is threefold:

1. The type system checks that structural types' declarations are valid, and that their uses conform to the type discipline of the program.

2. When Scala types are "erased" to Java virtual machine types, structural types disappear. All structural method calls are no longer valid in the erased type discipline. They are replaced by a special "apply dynamic" method call.

3. Every "apply dynamic" call is compiled to Java reflection so as to behave as a proper call on a structural value. The necessary caching infrastructure is added.

Type Checking In Scala, it is straightforward to support type checking of structural types: structural types are a generalisation of refinement types, which have been part of the language from the start. Like structural types, refinement types allow to add constraints on the members of an existing type. However, they require the constraints to be on the type of a member that is inherited: no new members may be defined structurally. To support structural types, the Scala type checker merely had to see a single test removed.

Erasure In order to understand why structural types have to be "erased", and why some method calls become invalid afterwards, it is necessary to discuss the role of erasure in the Scala compiler. Java bytecode is annotated with Java types: the virtual machine will check these types and will reject the program if they are not valid. A Scala program compiled to bytecode must be annotated with Java types that are valid and will not be rejected by the virtual machine. Erased types are Java types that define a *type discipline* on the program that is *equivalent* to, although less precise than that of the program typed with Scala types. If erasure maintains this equivalence, bytecode instructions—most importantly method calls—can be directly used to compile Scala programs: a method call which is valid in Scala is also valid in bytecode. Erasure is described in detail in §3.6 of the Scala specification [66].

Before structural types were introduced, all of Scala's type constructs could be erased in some way. To simplify, a mixin type such as "`A with B`" is erased to `A`—all objects of this type must then inherit `A` and implement `B` so that they can be cast to `B` when necessary. However, there is no Java type to which a structural types can be erased. The simple structural type "`Object { def f: Int }`", if erased to `Object`, will see the JVM reject a call to `f` because `Object` has no such method. A solution is to generate for every structural type in the program an interface that represents it and have all of the objects of this structural type implement the interface. The interface for the type above may be as follows:

```
interface FStruct {
  int f();
}
```

That is, in a simplified form, the "generative" technique used by Whiteoak. Scala's "reflective" technique does not require an erasure model that maintains the type equivalence property. Instead, the compiler will have to generate instructions that allow calling methods on objects having an erased type that does not contain the definition of the method.

"Apply dynamic" calls Method calls are represented in Scala's abstract syntax tree as `Apply` nodes. Normally, these nodes are compiled to JVM method call instructions; that is possible because of erasure, as we explained in the previous paragraph. During erasure, when the compiler encounters in the abstract syntax tree an `Apply` node that cannot be compiled to a JVM method call because the receiver has a structural type, it will replace the `Apply` with an `ApplyDynamic`.

The observed semantics of `ApplyDynamic` are that of `Apply`, but it is compiled so that the receiver of the call can have an erased type that does not define the method, as will be explained below.

As a reminder, the dispatching semantics of method calls in Scala, like in most popular object-oriented languages, is dynamic on the receiver's type but static on the parameters' types. Consider for example the method call "`a.f{x}`". The actual instances used at run time for `a` and `x` may have types that are compatible with, but different from the types statically assigned to `a` and `x`. The implementation of `f` that should be used is that defined in the class corresponding to `a`'s dynamic type. If, for a given call site, the type of the receiver instance changes, the implementation of the method must be modified. On the other hand, if `f` is overloaded, the alternative that should be used is statically defined, based on the static types of `f`'s parameters.

It may be noted that we present `ApplyDynamic` as a means to compile structural types. However, we believe that it is a more general solution that can be used for other types that bypass the JVM's type system.

Compilation of "Apply Dynamic" Calls

The technique that we present to compile `ApplyDynamic` nodes to JVM bytecode is based on Java reflection. At first sight, this transformation is trivial. Since reflective method lookup and application are purely dynamic, the JVM's type system won't be in the way. A method call of the form "`a.f(b, c)`"—where `a` is of a structural type, while `b` and `c` are of type `B` and `C` respectively—can be replaced by the compiler with the following expression.

```
a.getClass
  .getMethod("f", Array(classOf[B], classOf[C]))
  .invoke(a, Array(b, c)
```

In this implementation, the semantics of dispatching is preserved, mostly thanks to the fact that Java reflection supports it directly. The `getMethod` call is sent to `a`'s dynamic class, routing the lookup to the right place. The static types of the parameters, obtained with `classOf` operators, are sent to `getMethod` in order to select from overloaded alternatives.

Caches This naive compilation technique cannot be considered because its performance is not acceptable in practice; a purely reflective method call is about 7 times slower than a regular call (on JVM 1.6). However, about $4/5$ of the compiled expression's complexity lies in the

sole `getMethod` operation. If the method is somehow already available and only `invoke` and `getClass` are used, the reflective implementation is "optimal" and a method call may be only two times slower than a regular call.

Caching brings the performance of structural method calls closer to the optimal performance, when compared to the naive implementation. Of course, it never allows for an optimal performance since there will always be cache misses—at least once for the first call. We will discuss compilation techniques using two different caching strategies, which approach the optimal situation quite closely in realistic cases. In this discussion, and in §3.3 where the performance of caching is assessed, we will use the following caching techniques.

0c No caching. This is a slight variant of the naive, purely reflective, compilation techniques above, with the array of static parameter types precalculated.

1c Monomorphic inline caching is a technique where only a single method is cached for every call site.

Nc Polymorphic inline caching is a technique that caches a method for every receiver type at the call site.

Monomorphic caching A call site is monomorphic if the type of the method call's receiver stays the same throughout the execution of the program. Many call sites are monomorphic in practice. Monomorphic inline caching takes advantage of this property and caches one method implementation per call site. If the call site is in fact monomorphic, the same implementation can be reused every time at practically no cost. If the site is not monomorphic, the cached implementation will be discarded every time the receiver type changes, and a new implementation will be obtained using `getMethod`.

For every dynamic call site of a class, the 1c implementation adds a static method called `dynMethod`—plus an identifier to make that name unique. `dynMethod` takes the class of the receiver and returns a method implementation that corresponds to the call site and to the receiver. A call site of the form "`a.f(b, c)`" is replaced by the compiler with the following expression.

```
dynMethod(a.getClass).invoke(a, Array(b, c))
```

We wish to generate a `dynMethod` that can, for monomorphic call sites, return immediately. Here is how `dynMethod` is implemented for the above call site.

```
[static]
def dynMethod(forReceiver: JClass[_]): JMethod = {
  if (dynMethod$Class != forReceiver) {
    dynMethod$Method =
      forReceiver.getMethod("f",
        Array(classOf[B], classOf[C])
      )
    dynMethod$Class = forReceiver
  }
  dynMethod$Method
}
```

The static variables `dynMethod$Class` and `dynMethod$Method` contain, respectively, the class of the previous receiver and the looked-up method for that class. Whenever the previous receiver class differs from the current one, `dynMethod$Class` and `dynMethod$Method` are recalculated, at considerable expense of time. If they are equal, `dynMethod$Method` is returned, at the expense of a test and a variable dereference. The latter case is almost equivalent to immediately returning, and gives to the call site a performance close to the optimal.

Polymorphic caching A call site is polymorphic if the method call's receiver type changes over the lifetime of the program. Polymorphism has many dimensions: its degree—bimorphic, trimorphic, etc.—describes how many different receiver types can be observed at that call site throughout the program's execution; its intensity describes how frequently the call's receiver type changes. A call site may have a high degree of polymorphism but may remain quasi-monomorphic for most of the program's execution, for example if polymorphism at that point is linked to initialisation. It is, for obvious reasons, on highly intensive polymorphic call sites that it is the hardest to obtain a performance close to that of the optimal compilation. In what follows, and in §3.3, we discuss worst case polymorphic call sites; reality will usually lie somewhere between the worst case and a monomorphic site.

The technique that we use for polymorphic inline caching is based on that proposed in [41], using JVM objects instead of low-level memory blocks.

The technique generates a `dynMethod` method for every call site, and the call site itself refers to that method. Instead of caching a single pair made of the receiver's type and the method's implementation, `dynMethod` caches a list thereof. When looking-up the implementation of the method at a call site, and if the receiver type was never

encountered before, `dynMethod` will use `getMethod`, as in the naive implementation. The receiver class and method implementation pair will be appended to the front of the list. A method's implementation already in the list will be reverted from the list.

The list used to store method implementations is a simple linked list. Searching it has a complexity of $O(n)$ (n being the length of the list), adding to it has a complexity of $O(1)$. The latter task is a lot less common than the former; using a linked list may not be optimal when compared with a binary search tree, for example. We did not attempt to evaluate the performance of a cache backed by a binary tree or using a hashed dictionary. It must be noted that, contrary to most faster data structures, some implementations of linked lists can be used as caches in a multithreaded environment without synchronisation. In a parallel environment, a conflict leads at worst to one implementation obtained by `getMethod` being lost; it will have to be recalculated the next time it is used.

Exception handling When an exception is thrown in a reflectively called method, `invoke` wraps it in an `InvocationTargetException`. That changes the semantics of method calls. The problem is trivially circumvented by catching the exception from the `ApplyDynamic` call site, and then rethrowing the original exception, which can be obtained from the `InvocationTargetException`.

Boxing of native values Java reflection only works with objects, not native values like `int`, `float`, or `boolean`. If a method requires a argument of type `int`, for example, reflectively calling `invoke` on it will require a boxed integer (instance of `java.lang.Integer`). Similarly, if a method returns a `boolean`, for example, the result of the reflective call will be a boxed boolean (instance of `java.lang.Boolean`). The compilation of `ApplyDynamic` must take that into account.

The Scala language has a purely object-oriented unified type system with no distinction between native values and objects. `Any` is a type that is compatible with both native values and objects. A type variable (with an implicit `Any` upper bound) can be instantiated to either an object type or a native value type. Therefore, both `Any` and type variables must be erased to a type that the JVM recognises as compatible with all instances. `Object` is the closest approximation of such a type. However, using `Object` requires all native values to be boxed when they are referred to with a type that exceeds Java's `Object` in generality. Early versions of Scala used a custom boxing scheme that was incompatible with Java reflection. The compilation

of `ApplyDynamic` required constant unboxing of Scala boxed values and reboxing to Java ones, and vice-versa. The performance lost because of that was very noticeable. Changing Scala to use the same boxing technique as Java solved that problem.

Native bytecode operations Another difficulty caused by Scala's unification of types is that operations like integer addition (+) on native values, for which the JVM uses bytecode instructions, are represented in Scala as methods. The normal compilation has the bytecode generator recognise these "fake" method calls, and rewrite them to the corresponding instructions. Let us consider a program where a native value is referred to as a structural type, as in the example below.

```
def g(x: Any{ def + (i: Int): Int }) = x + 2
```

From Scala's perspective, `x + 2` is considered as `x.+(2)`. Since `+` is a structurally defined method, it must be compiled as `ApplyDynamic`. The `getMethod` call will lookup a `+` method on `x`'s class and will fail—`x` is boxed to a `java.lang.Integer`, which does not have a `+` method.

Instead, the `ApplyDynamic` transformation detects any fake method —using a list of such methods—and generates a call to a static utility method that implements the fake method's behaviour. Of course, a fake method like `+` may actually be dispatched to an instance that does, in fact, implement it. Therefore, the call to the utility method for the addition must be guarded by a dynamic type check: if the call's receiver is of type `java.lang.Integer`, the utility method should be used; if not, the call must be dispatched like a normal `ApplyDynamic`.

Type parameters To call a method, `invoke` must know the static types of the method's parameters, which are passed as an array of classes. These types are used to maintain correct dispatch semantics and to select the right method if its name is overloaded. When `invoke` is used to implement an `ApplyDynamic`, the compiler must know the signature of the structurally defined method that is being called. In general that is easy, as in the example below.

```
def g(x: { def f(a: Int, b: List[Int]): Int }) =
  x.f(4, List(1,2,3))
```

The declaration of the structural type of `x` contains the static types of both parameters of `f`. These types can be used to generated the reflective call to `invoke`.

There is no problem either if the parameter types of the structurally defined method are type variables, as long as the variables are declared as part of the method definition.

```
def g(x: { def f[T](a: T): Int }) =
  x.f[Int](4)
```

When g is called, its argument p will be an object that contains a method with the signature "x[T](t: T)". Erasure changes type variables to their erased upper bound, which is Object in T's case. Erasure will make sure that f's parameter is boxed when it is a native value. This is the expected behaviour in such a situation: invoke is called with Object as the static type for the a parameter.

A problem arises if the parameter types of the structurally defined method are type variables that are declared outside of the scope of the structural type.

```
def g[T](x: { def f(a: T): Boolean }, t: T) =
  x.f(x.t)
g[Int](new { def f(a: Int) = true }, 4)
g[Any](new { def f(a: Any) = true }, 4)
```

The type variable T is instantiated to a concrete type every time g is called. The static type of f's parameters therefore changes for every call to g. On the other hand, the transformation of ApplyDynamic for x.f is done only once, in the body of g, no matter what type T will eventually be assigned to. The values of type variables are not available at runtime so that ApplyDynamic cannot be compiled in a way that reconstructs the static types of the method's parameters at runtime.

A similar issue arises if the parameter types of the structurally defined method are type variables that are declared as type members of the structural type.

```
def g(x: { type T ; def t: T ; def f(a: T): Boolean }) =
  x.f(x.t)
g(new { type T = Int; def t = 4; def f(a:T) = true })
g(new { type T = Any; def t = 4; def f(a:T) = true })
```

As before, the ApplyDynamic for x.f cannot be compiled because it depends on the static type of T. In this situation, T changes every time the parameter x of g implements the structural type using another value for type T.

To overcome these problems requires runtime types that complete the missing static type information. Manifests, which are optional runtime types and will be presented in Chapter 6, could be used so

that the values of type parameters are available to the implementation of `ApplyDynamic`. However, this remains future work; for the time being, Scala rejects problematic type variables for structurally defined method's parameters.

3.3 On performance

All the tests of this section have been run using a Java HotSpot 1.6.0_07 64-bit Server VM on an iMac computer (2.33 GHz Core 2 Duo with 4MB L2, 2GB Memory) running Mac OS X 10.5.6. All times are averages over multiple executions, and their variances have been checked to be insignificant—typically in the order of $1/100$ of the average. For tests involving pseudo-random values, values change between executions.

Dynamic and virtual method calls

Only a fraction of method calls will be structural in real-life scenarios. The first performance benchmark that we discuss is a functional implementation of merge sort, which uses structural types. Such code is representative of the performance of code that heavily depends on structural types.

Functional merge sort To implement a functional merge sort algorithm for a list, a comparison operator between elements of the list is needed. On solution is for all objects that are to be merge sorted to implement the `Comparable` trait in Scala's library. But if merge sorting is to happen on objects that are not ready for it—that is, have not implemented `Comparable`—structural types can be used instead. This is the signature of the `mergeSort` method when using an interface:

```
type ComparableList =
  List[Comparable[Any]]
def mergeSort(elems: ComparableList): ComparableList
```

When using structural types, the signature of the method remains identical; only the type is declared differently:

```
type ComparableList =
  List[{ def compareTo(that: Any): Int }]
```

The implementations of both methods are identical. It requires $O(n \log n)$ merges on average. Each merge is composed of three calls for dereferencing list heads or tails, one comparison (a call on a structurally defined method), and one concatenation (which instantiates an object).

Figure 3.1 charts the time required to order 2000 times a list of 1000 pseudo-random elements. For *monomorphic* measurements, the list contains only instances of a single class. That means that the structurally-defined comparison method can always be called on the same implementation. For *bimorphic* measurements, the list contains instance of two classes at even, respectively odd positions. That forces the implementation of the method to be changed whenever the receiver object of the comparison method changes. For both monomorphic and bimorphic lists, the performance of different caching techniques is measured. S is the reference situation where a `Comparable` interface is used instead of structural types, and where regular interface calls are used instead of `ApplyDynamic` calls.

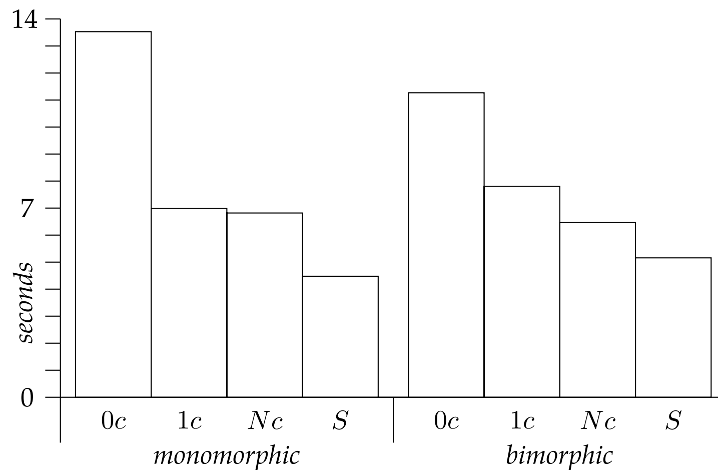


Figure 3.1: Execution time of the merge sort algorithm. $0c$ is the uncached implementation, $1c$ uses a monomorphic cache and Nc a polymorphic one. S uses interfaces instead of structural types.

As expected, there is a performance penalty for using structural types when compared to using a `Comparable` interface; in this case the time is around 25% more for Nc caching. If we assume that uncached structural method calls are about 10 times slower than regular interface calls, our data indicates that calling the comparison method represents about $1/8$ of the test's execution time in the reference case.

In the monomorphic case, Nc and 1c exhibit similar performances, while 0c's performance is, predictably, more than twice as slow as regular interface calls. In the bimorphic case, the advantage of Nc becomes evident. That is not surprising as, in the test procedure, 1c must revert to a full method lookup equivalent to 0c for half of the calls. Indeed, the receiver of the call is randomly chosen between the two same classes for each call, which therefore has a 50% chance of

being different from the previous one. The relative speed of Nc and 1c may vary under different scenarios, but no bimorphic scenario is more favourable to 1c than the monomorphic case. On the other hand, I cannot explain the difference in performance between the mono– and polymorphic cases for 0c.

Single method call The test repeatedly calls a structurally defined method in a tight loop. The degree of polymorphism at the call site varies from one to four—from mono (morphic) to quadri (morphic). Figure 3.2 charts the time required to call a method 10 million times. Bi- and quadrimorphic cases are worst-cases, where the class implemented by the receiver changes for every method call. To obtain polymorphism at the call site, the call's receiver is looked-up in an array, changing the index into the array, modulo the degree of polymorphism, for every iteration of the loop. Profiling this operation did not reveal the relative cost of this operation.

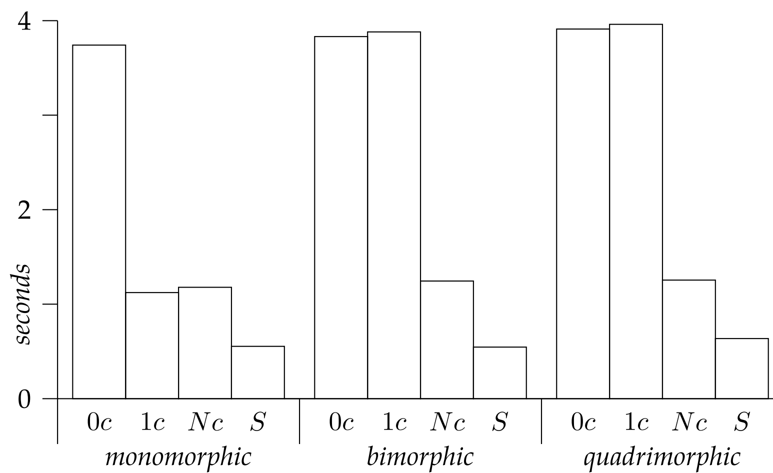


Figure 3.2: Execution time in relation to polymorphism degree of call site.

As was previously mentioned, the uncached implementation (0c) of `ApplyDynamic`, when compared with a regular interface call (S), is 7 times slower. When polymorphic inline caching is used (Nc), the slowdown is reduced to between 2.0 and 2.3 times, depending on the degree of polymorphism. The situation for monomorphic inline caching (1c) is equally favourable for a monomorphic call site, but the performance is as bad as 0c for polymorphic sites. The performances of the three caching strategies for different degrees of polymorphism are perfectly coherent with their implementations. If we assume that the time of a method call in Nc is entirely composed of the execution

of the `invoke` method, comparing Nc and 0c shows that the time of a full reflective method call is distributed between 30% in the `invoke` call and 70% in the `getMethod` call.

Figure 3.3 represents the same data as figure 3.2, but the JVM is run in such a way that it does not use just-in-time compilation, only interpreting the bytecode. While this data is not representative of real performance, it may be considered as an upper bound for the slowdown caused by structural types, and shows how effective JVM 1.6's just-in-time compilation is at optimising them. In the interpreted case, a regular method call is 7.6 times faster than for the Nc implementation, while when just-in-time compilation is used, the difference is of only 2.1 times.

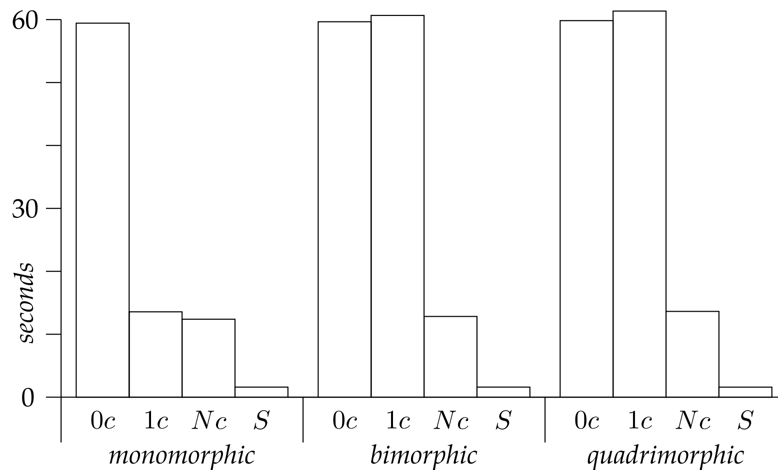


Figure 3.3: Execution time—with JVM just-in-time compilation disabled (interpreted)—in relation to the polymorphism degree of call site.

Megamorphic call sites Some call sites have a degree of polymorphism much higher than that used in the previous test. Such sites are rare—Hölzle et al. [41] report that no call site in their real-life benchmarks have a degree of polymorphism of more than 10—but they can seriously harm the performance of polymorphic inline caching. Figure 3.4 graphs the time (y-axis) required to call a method 10 million times with respect to the degree of polymorphism of the call site (x-axis).

In Nc, the length of the cache n is eventually equal to the degree of polymorphism of the call site. Searching the cache has a complexity of $O(n)$ so that the worst-case performance of Nc will decrease linearly with the degree of polymorphism. On the other

hand, the performance of 0c is expected not to change with the degree of polymorphism. The results do show a slowdown for the latter case, possibly because of variations in the effectiveness of just-in-time compilation. However, this slowdown remains lower to that of Nc so that the performance of the two implementations become equal when the call sites have a degree of polymorphism equal to about 180. At that point, the performance of 0c and Nc are equivalent because Scala's implementation of Nc automatically reverts to 0c when reaching that threshold.

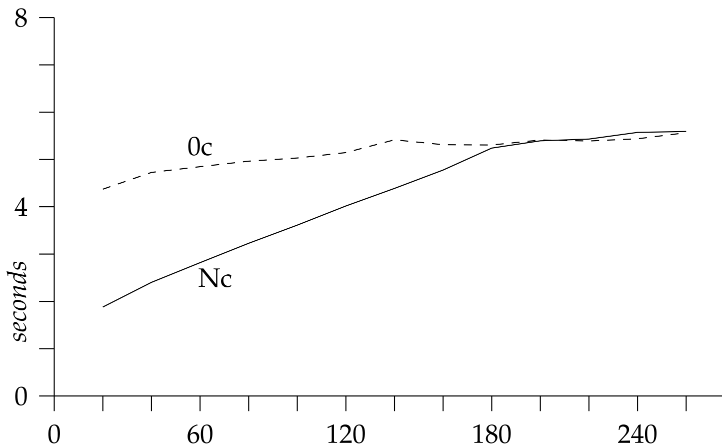


Figure 3.4: Execution time in relation to the degree of polymorphism of the call site.

Interpretation The results reported above show that the Nc implementation, using polymorphic inline caching, is either significantly superior or roughly equivalent to the other implementations in all situations. Even for monomorphic call sites, the 1c implementation that is designed with such call sites in mind is not significantly better than Nc.

Scala and Whiteoak

Gil and Maman have compared [38] the performance of structural method calls in Whiteoak with the performance of regular interface calls. Their results show that in the best case—when the fast "primary cache" can be used—their implementation allows for constant-time structural method calls that are about as fast as interface calls. However, a start-up cost of about 200μs is incurred before the first structural call is executed. They also report that the worst case scenario is about 7 times slower than the best case. Sufficiently small

performance-critical areas are covered by the primary cache and will be close to the best-case performance. For most real-life applications, however, that would rarely be the case.

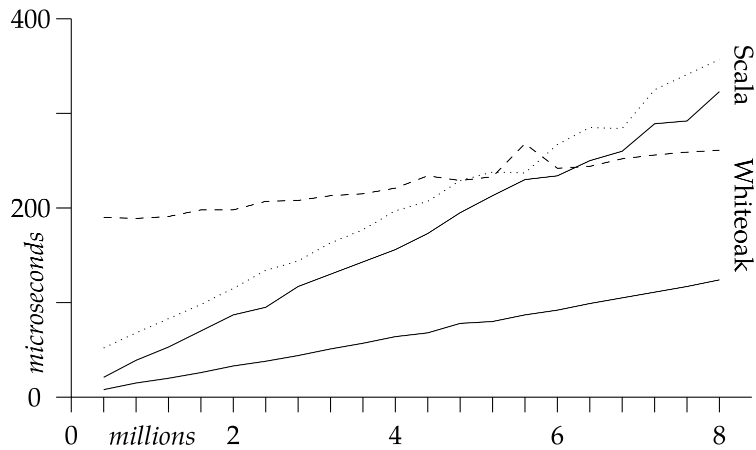


Figure 3.5: Execution time in relation to the number of monomorphic calls at single call site. Broken lines are for cold call sites.

Figure 3.5 shows the results obtained in our own tests, comparing the performance of Whiteoak's best case with that of Scala's Nc implementation for a monomorphic call site. Results for polymorphic call sites do not differ significantly. The observed times are linear in the number of calls: they confirm the constant call times reported by Gil and Maman. To understand better the initial start-up cost of Whiteoak, the tests are run twice, without restarting the JVM, for different call sites. The reported "cold" times are for the first structural call site, and include the infrastructure's initial start up overhead. The "hot" times are for the second call site, and only include the overhead incurred at a given call site, if any. A best-fitting linear regression ($y = ax + b$) is calculated for every data set. The value at which the regression line intercepts the y-axis (b) is an approximation of the overhead; the slope of the line (a) is an approximation of the time for a single structural call.

	Overhead	Call time
Cold Scala	36μs	38ηs
Scala	3–6μs	38ηs
Cold Whiteoak	180μs	11ηs
Whiteoak	2μs	15ηs

A comparison of hot Scala with hot Whiteoak shows that both have relatively low overheads when calling a site for the first time—equal

to about 80,000–160,000 method calls for Scala, 130,000 for Whiteoak. The call time for Scala is about 2.5 times slower than that of Whiteoak, which is in line with the result reported above. The initial start-up overhead of Scala's implementation is one order of magnitude greater than the overhead at a subsequent call site. This overhead's origin is likely to come from the infrastructure for polymorphic inline caching, as this is the only code that is shared amongst call sites. The initial start-up overhead for Whiteoak's implementation is 5 times greater than that of Scala, and can probably be explained in part by the time necessary to initialise ASM, the bytecode manipulation framework required by Whiteoak. Scala's call times are 3.4 times slower than cold Whiteoak's call times, which is a result at odds with those comparing Scala's times with regular interface calls. A possible explanation is that just-in-time compilation differs between this test and those obtained on Scala alone.

Interpretation The results that are reported above show Whiteoak's implementation of structural types to be superior to Scala's implementation. They indicate that Scala is in between 2.5 and 3.5 times slower than Whiteoak when considering the performance in tight loops, a situation where Whiteoak excels. Therefore, Whiteoak's implementation is preferable if structural method calls are to be used in a hot spot of a performance-critical algorithms. On the other hand, these results very much depend on Whiteoak using its small primary cache. Since Whiteoak's cache is shared amongst all call sites—contrary to Scala's caches which are specific to each call site—the performance of Whiteoak's implementation will diminish when a program's execution cost is not contained in a tight loop. Gil and Maman report the performance of Whiteoak in a situation that "mostly hits the secondary cache" to be 7 times smaller than the best-case performance. Because of that, in such a situation, Scala's technique using reflection and polymorphic inline caching will likely be faster.

There is a performance penalty associated with the use of structural types on the JVM, but a reflective compilation technique that uses efficient caching brings this cost to acceptable levels for all but the most critical programs. Generative compilation techniques such as that of Whiteoak have, in principle, the potential for even higher performance. On very tight loops, a situation that may be relevant to certain performance-critical algorithms, generative techniques are clearly superior. On the other hand, reflective implementations are likely to yield superior performance when the cost of using structural

types is distributed throughout the program. Structural types should in any case not feature prominently in tight, performance-critical loops on the Java Virtual Machine. Even though Whiteoak can provide almost native performance, the requirements to allow it are stringent, and any deviation immediately leads to very poor results. Therefore, reflective compilation techniques are likely to lead to higher real-life performance.

From the point of view of the implementation, reflective techniques are also simpler. They only require changing the compilation scheme for structural method calls and a small runtime library that implements polymorphic inline caching. Generative techniques, on the other hand, require changing the compiler to generate interfaces for each structural type, changing the compilation scheme for structural method calls—to adapt the receiver to the structural type using a wrapper—and require an infrastructure to generate wrapper classes dynamically and to cache them. A dependency on a code generation library is added to all programs. Furthermore, loading generated classes at runtime requires access to the program's class loader, which may not be available when running on environments such as application servers.

There is only one language restriction in the current implementation: the limitations on type variables in Scala's structural types. As was mentioned before, using dynamic types such as those provided by manifests is likely to lift this restriction. Gil and Maman's article does not detail the restrictions that the generative technique imposes upon structural types, although Whiteoak does not allow "generic structural types", a restriction that is similar to that imposed on Scala's type variables. We can assume that Whiteoak would have similar difficulties with type variables in structural types. Other important features, such as structural arrays, are not available with Whiteoak's generative technique. Whether generative compilations techniques could be used to compile Scala's structural types, which are part of a much more complex type system and language than those of Java, remains an open question.

Reflective compilation techniques are likely not to be noticeably slower than generative ones in real applications. However, reflective techniques are simpler to implement, have fewer dependencies and have been shown not to restrict the language. In this context, it is worth noting that a recent overhaul of Whiteoak's source (version 2.1) did not retain the generative compilation infrastructure, leaving it to be implemented later, and uses a simple un-cached reflective implementation.

3.4 Type-safe relational algebra

Let us now step back from the implementation and performance of structural types, and consider their use in domain-specific programming. To do so, this section discusses the example of a type-safe relational algebra library [59], which was implemented by Cédric Lavanchy, under my direction. The relational algebra domain-specific language provided by the library is automatically converted into SQL strings. However, this second function of the library goes beyond the scope of this discussion. As was mentioned in the introduction of this chapter, a relational algebra is notoriously difficult to represent in object-oriented type systems. Many popular libraries, such as JDBC, do not even try to integrate the domain within the host language. So called object-relational systems use a simple mapping that represents relations as classes. While this allows domain-specific soundness properties to be checked by the host's type system, it remains unsatisfactory because it restricts the algebra's ability to join relations.

Lavanchy's library explores solutions to write type-safe queries on databases using a relational algebra. To do so, it uses structural types and compound types (see §3.2.7 of [66]), which better correspond to the properties of a relational algebra than object types, as we will see below. Of course, because databases can be modified in parallel to the programs that query them, the notion of type safety is not absolute. Instead, it is relative to a given representation of the database, following the schema below.

- The program contains a static representation of the structure of the database. This may not be complete but describes the minimum requirement that are needed by the program.

- Queries against the database are statically checked to be compatible with its representation. When reading data from the queries—that is, when converting domain-specific data to host values—types are maintained and checked.

- When running the program, its assumptions about the structure of the database are checked à priori, so that a database of an invalid type is detected during initialisation.

Full typing extends to the relational algebra operators and to results. However, Lavanchy's library is very much a prototype, lacking crucial features. We will see below that some aspects of the domain-specific language remain untyped—most notably boolean expressions used by the *select* operator. Also, we will see that, while the library is correct

in terms of typing, the language is missing features that prevent its implementation to return values fitting the types. The flaws of the library will lead to a discussing about what these limitations demonstrate in term of existing domain-specific programming support.

The following program is an example of the library's use, whose implementation we will discuss in the rest of this chapter. The first snippet defines the structure of the database as a set of structural types. The reason for using structural types will become clear below. Only its fraction that is relevant to the program is represented; the database may contain many more elements which the program will simply ignore.

```
type Person = {
  def personId: Int
  def firstName: String
  def lastName: String
}
type Attends = {
  def lessonName: String
  def personId: Int
}
type Lesson = {
  def lessonName: String
  def teacherId: Int
}
```

The data encoded by this database concerns teachers teaching lessons to students. A person can either be a teacher or a student. In the former case, he is linked to the lesson he teaches through the `Lesson` relation; its `personId` represents the teacher. There is only a single teacher for a lesson, but he can teach multiple lessons. In the case of students, it is the `Attends` relation that defines the link, allowing multiple students to take multiple lessons. The user of shared identifiers to associate values is typical of relational data, and clearly demonstrates that this is not object-oriented data.

The next snippet declares that the three types above are actually tables in the database.

```
val person = table[Person]
val attends = table[Attends]
val lesson = table[Lesson]
```

A table is a base relation that is directly stored in the system; other relations are obtained by composition using relational operators. A relation is a set of disjoint tuples, with all tuples the same size and

containing values of the same type. Therefore, `table` requires a type parameter defining the structure of the tuples in the relation. This is the type that will be maintained when using relational operators to create new relations. Tables are given names in the Scala program, so that they can be called later.

The next snippet declares a method that queries the database, using the relational algebra. It returns the database identifier for a person of a given name.

```
def personId(name: String) =
  unique(person select { _.lastName == name }).personId
```

Of all tuples in table `person`, the *select* relational operator only keeps those which `lastName` field is that being searched for. The unique method receives the resulting relation and checks that it contains a single tuple. If it is the case, it returns the tuple, discarding the relation; otherwise an error is returned. The select operator does not change the type of the relation.

```
class Relation[+T] {
  def select(predicate): Relation[T]
  ...
}
```

The type of the relation received by unique therefore remains `Person`. Scala's type system can guarantee that reading its `personId` is safe and can infer its type as being `Int`. However, until now, the type guarantees offered by Lavanchy's library do not differ from those of an object-relational mapping.

In the next snippet, we will see how structural and compound types allow us to type relational operators that wouldn't be allowed using objects. The query defined below finds the names of all students of Mrs Hopkins.

```
def hopkinsPupils =
  project[{
    def firstName: String;
    def lastName: String
  }] {
    (person join attends join lesson) select { r =>
      r.teacherId == personId("Hopkins")
    }
  }
```

This example utilises two new relational algebra operators: *join* and *project*.

The join operator Join takes two relations to create a new one by combining all tuples of the first with all tuples of the second. If the combined tuples contain fields of the same name, they will only be joined if they are equal, following the natural join semantics. As an example, consider the following two relations.

person

personId	firstName	lastName
1	Barak	Simon
2	Leïla	Kaltenbacher
3	Karmen	Michelet
4	Sébastien	Möhn

attends

lessonName	personId
Math	1
Math	2
Math	4
French	1
French	3

The result of joining these yields the following relation.

personId	firstName	lastName	lessonName
1	Barak	Simon	Math
1	Barak	Simon	French
2	Leïla	Kaltenbacher	Math
3	Karmen	Michelet	French
4	Sébastien	Möhn	Math

The type of the tuples in this relations does not correspond to any that was originally declared. In an object-relational mapping system, no class would be available to type that result. However, Scala's compound types allow to define a correct type as a mixin between `Person` and `Attends`.

```
class Relation[+T] {
  def join[U](that: Relation[U]): Relation[T with U]
  ...
}
```

To go back to the example program, the tuples in the relation resulting of the triple join is a compound type with the following members.

```
{
  def personId: Int;
  def firstName: String;
  def lastName: String;
  def lessonName: String;
  def teacherId: Int
}
```

The project operator When joining relations, not all fields of the tuples may be necessary. Many may be irrelevant to the query, or only used internally. A projection keeps a subset of the fields of the original relation's tuples. For example, projecting the joined relation above on the fields `lastName` and `lessonName` yields the following, new relation.

`lastName`	`lessonName`
Simon	Math
Simon	French
Kaltenbacher	Math
Michelet	French
Möhn	Math

As was the case with joins, the type of the relation resulting from the projection is none of the three types that were explicitly declared. Furthermore, the projection type can only have members which name and type exist in the original relation's tuples. This property cannot be represented using object types; the supertype relation of structural types, however, correspond to it. Therefore, if the projection type is declared as a structural type, as is the case in the example program, the typing constraints for the project operator can be defined correctly.

```
class Relation[+T] {
  def project[U >: T]): Relation[U]
  ...
}
```

A compilation error will be generated if the user of the library projects a relation onto a type that isn't compatible.

The final snippet uses the relation resulting from the query to print a list of students.

```
for (p <- query(hopkinsPupils)) {
  println(p.firstName + " " + p.lastName)
}
```

This code is outside the domain-specific fragment, and uses a standard Scala for-loop to iterate over the tuples of the relation. However,

because the domain-specific fragment is assigned types of the host language, the data it produces integrates perfectly. In this example, the type of `p` is the structural type resulting from the projection above.

```
{
  def firstName: String;
  def lastName: String;
}
```

The compiler can check that the two calls to read the pupil's name are indeed present on instance `p`, and that their results are strings.

The rename operator The example above demonstrates that compound and structural types allow to satisfactorily type complex relational operators. In fact, all but one relational operator can be typed using similar rules to those described above. The operator that can't be typed is the rename operator, where a given field is renamed to another name in all tuples. For example, if the relation resulting from the `hopkinsPupils` query has its `firstName` field renamed to `givenName`, tuples in the new relation have the following type.

```
{
  def givenName: String;
  def lastName: String;
}
```

Such an operation at the level of types is impossible using nominal types as well as using structural types.

Ay, there's the rub!

The impossibility of typing the rename operator is a minor nuisance—alternatives exist, albeit clumsy ones. Two more pressing issues need to be considered if the type-safe relational algebra library is to function.

Instances of types This library must create instances of compound and structural types when executing certain queries. For example, each tuple of the relation returned by executing the `hopkinsPupils` query is an instance of the following structural type.

```
{
  def firstName: String;
  def lastName: String;
}
```

As must be clear from §3.2, such types do not exist in the Java virtual machine, and cannot be instantiated. Calling methods on structural types requires a complex reflective infrastructure, and an even more complex system would be needed to instantiate them.

Should the Scala compiler be extended with a complex static and runtime infrastructure that allows instantiating such types? Probably not because when using structural types without a domain-specific focus, there is no reason to create instances for them. Values exist at runtime quite independently of the type that the compiler used to represent them. For example, structural types would be used to represent values that do not share a common class, but share a common method—for example `close`. However, typing values using a structural type or explicitly as classes does not imply a runtime change to the values themselves.

In fact, instantiating structural types seems suspect in a class-based language. There, class types represent an explicitly defined set of values: every call to the class' constructor adds a new element to the set. Structural types are not defined by listing their elements like classes. Instead, they are like set operators that are defined by filtering other sets. Because instantiating a type is adding an element to its set, types that are not defined as sets but as operations cannot logically be instantiated.

A similar problem exists for compound types, where the **`with`** operator on types has no corresponding operation on values. Using compound types, it is possible to take two classes and create a new type from the union of their members. It is not possible to take two values and create a new value by uniting their members.

Furthermore, it is practically impossible to give a meaningful semantics to instantiating structural types or merging two values into a compound. Structural types are pure definitions, without an implementation, which would somehow need to be chosen arbitrarily. Compound values suffer from the fact that the same member may be present in both values. There is a well-defined semantics for this in Scala, based on linearisation. However it is delicate when the linearisation is declared explicitly and would be all but unmanageable when compound types are inferred, as is the case in the relational algebra library.

In domain-specific programs that use complex types for soundness, it may nevertheless be necessary to create such structural or compound types instances. This is because domain-specific types may not simply represent values created by the program, but also values created by domain-specific code. In this case, the link between

values and types is not natural but must be constructed explicitly. For example, in the relational algebra library, queries have a type that corresponds to the expected form of the tuples they can return, Upon execution, they are further transformed into SQL strings, losing any natural connection to their type that may have been present in the structure of the query. The result of the query is known to represent data that is compatible with the type of the relation, but its actual form—a table of raw, untyped data—has lost its connection to the type system. The domain-specific library must reconstruct that link by creating instances of the expected tuple type. Of course, in any but the simplest cases, this type will be a complex structural or compound type.

In the relational algebra library, the problem of defining a semantics for instantiating structural or compound types is solved by domain-specific knowledge. Members of structural types are known to correspond to fields in the result set. When two values are united in a compound, the rules of the relational algebra guarantee that any duplicate members correspond to the same field in the result set.

Expressions In the example of this chapter, the predicates passed to the select operator where expressed as anonymous functions, and strongly typed, as in the example below.

```
person select { p => p.lastName == "Assange" }
```

However, here again, the type-safe design does not withstand the test of implementation. Lavanchy's library requires select predicates to be provided as SQL strings, like JDBC.

```
person select { "lastName = 'Assange'" }
```

The reason for this is that the backend of the library must convert relational algebra expressions into SQL strings to be shipped to the database. The various operators are accessible to the library because they are in fact constructors for a data structure that represents the relational expression. On the other hand, the predicate is simply an arbitrary, boolean Scala expression. Of course, a technique like ZyTyG could be used to parse domain-specific predicates. However, as was thoroughly discussed in Chapter 2, this would be only marginally more type-safe than using strings. In any case, creating a new domain-specific syntax for select predicates would be counterproductive, as one of the library's goal is precisely to allow relational data to be accessed using Scala types and concepts. In other words, the embedding domain for a relational algebra library must be limited to correctness, semantics and data.

3.5 Structural types for domain-specific programming

At the beginning of this chapter, I supported our interest in structural types by the additional flexibility they may offer to domain-specific programming. Clearly, the typing paradigm they represent is fundamentally different from other types. So much so that, contrary to other of Scala's complex types, structural types cannot be reduced to the object-oriented types of the Java virtual machine. Lavanchy's type-safe relational algebra library shows that, in some cases, they do indeed offer a flexibility that object types do not. But structural types match perfectly with the typing discipline of relations—in fact, relational types are a form of structural types. Is it that a relational algebra is a particular case that happen to benefit particularly from structural types? Or are relational algebras representative of a larger family of domains with similar type properties?

An informal poll of Scala developer concerning their use of structural types hinted to the following usage patterns. Most structural types serve as a way to assign an interface to code that is out of a programmer's control. For example, structural types allows for added flexibility when loading third-party modules by reflection, whose interface may be changing or may be inaccessible. This is obviously not an example of domain-specific programming. However, some clearly domain-specific uses of structural types were also reported. For example, one user mentioned how structural types may be useful to type configuration attributes for OpenGL shaders. There, the type and number of arguments differs depending on the shader's version and model. Structural types define the shader arguments required by the domain-specific program, and allow them to be composed. In fact, this example resembles in its basic structure what is done in the relational algebra library. First, the required structure of the tables or shaders is defined. Then, depending on the relational operations or shader code, these types are refined or joined.

We cannot at this point reach definitive conclusions about the role that structural types may play in domain-specific programming. However, the discussions in this chapter call for a more fundamental questioning about the relation between language abstractions used for domain-specific programming, and the implementation of domain-specific libraries. At the end of the previous section, we discussed two issues with the implementation of the type-safe relational algebra library.

1. Types available to the compiler are also needed by the runtime of the domain-specific library to implement its semantics.

2. Fragments of the original source code representing predicates are to be evaluated according to the domain-specific semantics implemented by the library.

These problems are independent of whether structural types are a suitable abstraction for domain-specific programming. Similar issues arise using different abstractions. To generalise, these problems have to do with the fact that a domain-specific language implemented as a library breaks the traditional barrier between the static and dynamic parts of the program. A fraction of the compilation—that related to domain-specific code—is outsourced to the runtime library. By that definition, a very similar issue is also raised in Chapter 2 concerning parsing of domain-specific syntax.

I believe that blurring of the traditionally clear separation between static and dynamic elements is a crucial characteristic of domain-specific programming. Aggressive use of general-purpose language abstractions in ZyTyG and structural types contribute to making domain-specific programming better. But both techniques succumb with the same symptoms: they require information that is part of the confined *static* domain, and that is inaccessible from their *dynamic* position. This is a fundamental problem that requires abstractions allowing library code to break down the walls of the compiler. The second part of this thesis picks up this assumption, to consider which such abstractions already exist, and what can make them better for domain-specific programming. As we will see, most of the issues raised in this and the previous chapters can be overcome by using the right *metaprogramming* abstractions.

At the end of this part, we have given substance to the first portion of the conjecture underlying this thesis. We have also seen what differentiates domain-specific libraries from other.

> Modern statically-typed object-oriented languages such as Scala have language abstractions that allow to satisfactorily host domain-specific programming. Domain-specific syntax, semantics, correctness properties and data can be provided without preprocessor or custom compiler. However, more static data must flow to the runtime implementation of domain-specific libraries than what is required by other libraries.

. . .

Part II

On Metaprogramming

Chapter 4

Controlling Code and Structure of Programs

The two embedding techniques for domain-specific languages that were described in Part I exemplify how one can utilise existing general-purpose constructs to encode embeddings as libraries. However, these examples also demonstrate the weakness of such techniques. When doing domain-specific programming, the strict separation between static and dynamic information is problematic. Some static knowledge is required by the runtime of domain-specific libraries.

The first example, in Chapter 2, described how to lift domain-specific fragments using the ZyTyG technique. Existing host language abstractions go far in terms of embedding an arbitrary syntax, and it is easy to integrate values of the environment into domain-specific fragments using implicit conversions. Indeed, a ZyTyG embedding is entirely defined statically; at runtime, values are created according to a pre-defined plan, like in any other program. There is no unusual interaction between the compiler and runtime. However, ZyTyG is poor when it comes to verifying correctness. In most cases, host language types cannot be used for defining syntax à la ZyTyG and simultaneously to provide other correctness properties. Alternatively, type checking could be done at runtime by specialised code in the library, when domain-specific expression are evaluated. Of course, this makes of domain-specific fragments second-class citizens, which are not checked statically. Still, this could be a good compromise in practice. However, with techniques like those we have discussed in Part I, even this can't be done. The problem is that a ZyTyG domain-specific fragment does not live in isolation: they refer to other values, methods, maybe other domain-specific fragments. The type information about these values and methods is not generally available

at runtime, as we will see in Chapter 6. One hits again the boundary between the compiler and runtime.

The second example, in Chapter 3, described how structural types can be used on data-based domain-specific languages, such as a relational algebra. Problems similar to those in the first example arise here too. The runtime evaluation of domain-specific fragments needs access to static type information. It isn't to check correctness, which is taken care of by structural types, but to implement the right semantics. Furthermore, this example also demonstrates how Scala expressions ought to be used to define the predicates of "select" statements, but cannot. This would require host language fragments inside domain-specific code to be accessible to the domain-specific library as code, not as reduced value. With either techniques that we have discussed in Part I, this cannot be done.

It appears that library-based domain-specific embedding techniques stand out through the way they straddle the dynamic and static portions of the program's execution. Both examples in Part I do not cover the whole embedding surface—the first missing correctness, the second semantics and syntax. Both are restricted because, to fill-in that missing area requires hooking into the compiler in a way that isn't possible using first-order programming abstractions. Domain-specific programming requires the compiler to leave traces for the runtime, and requires the runtime to have access to some of the compiler's logic. This situation certainly explains why domain-specific programming has long been tied to dynamically-typed languages with a small syntax, such as LISP, which do not enforce much of a separation between the analysis of the program and its evaluation.

On the other hand, we have seen in both examples above that domain-specific programming can greatly benefit from statically evaluated types—more precisely from abstractions that derive from it. ZyTyG benefits tremendously from implicit conversions, which require static types. The relational algebra library builds its correctness properties on the static type system. To take advantage of these benefits, and still reduce the separation between static and dynamic portions of the program, requires metaprogramming.

Metaprogramming has been used to describe a variety of techniques, some of which will be discussed below. The usual definition of "programming the program" is so vague as to be almost useless. This dissertation aims to provide a form of metaprogramming that allows the running program to observe and modify its state in order to adapt its domain-specific behaviours. It describes abstractions and facilities that create a tighter bond between runtime and compiler. It goes

beyond Java-like reflection, which only gives access to the program's interface and to its values. It is however not a fully dynamic form of programming, as it aims to stay within the constraints imposed by strongly typed virtual machines. It is a compromise that opens enough gaps in the wall between static and dynamic data to make domain-specific programming possible.

In the remainder of this chapter, we will discuss existing metaprogramming techniques. Some will serve as inspiration for the work in subsequent chapters. However, they differ in their focus, being general-purpose constructs and not targeted at domain-specific programming. By concentrating our efforts on metaprogramming for domain-specific concerns, these ideas can be simplified to better integrate with the design of Scala.

This does not imply that metaprogramming discussed in this dissertation is only useful for embedding domain-specific languages. In fact, as I hinted in the introduction to Part I, domain-specific programming is a broad concept that can be applied to many problems: extensible programs using plugins, application servers or other hosted applications, object-relational mapping frameworks, aspect-oriented programming, etc. Any problem that requires deeper changes than combining language abstractions in their most literal sense may be considered as domain-specific. For example, extensible programs require swapping in segments of code, and type-checking them. We have already discussed in Chapter 3 how a relational algebra is domain-specific. As for application servers, the multiplication of XML files describing the static structure of applications are a telltale sign that servers would require direct access to static information. It is no coincidence that all the applications that were just mentioned are usually heavily reliant on reflection. Indeed, I believe there is a very fundamental relation between reflection—one aspect of metaprogramming—and domain-specific programming.

The remaining two sections of this chapter analyse the state of the art of various relevant metaprogramming techniques. They discuss two major families of metaprogramming abstractions in statically-typed languages. In line with Scala's objective to integrate functional and object-oriented paradigms, the first has its intellectual origin in functional programming, the second in object-oriented. In §4.1, we discuss staged metaprogramming, which makes abstract syntax trees part of the program while maintaining static type safety guarantees. Mirror-based reflection rectifies some of the flaws of traditional reflection by enforcing certain design properties. It is described in §4.2.

4.1 *Related work:* staged code

For every language, there is a person who wants to see it extended by a new language construct. It is particularly true for language that originated with a very small set of features but nevertheless became popular, such as LISP. This may explain why metaprogramming tools to control language semantics where originally linked to this language. Smith's *reflective processor* [85] opens the way to staged metaprogramming. It describes the program's evaluation akin to an infinite stack of processors, each implementing that below. Reflective procedures in the program are evaluated as part of the current processor's implementation, that is, describing the semantics of the processor below. This technique was used to implement such semantically complex systems as the CLOS object system for LISP [4]. However, metaprogramming in LISP is that of dynamically-typed language. For example, the system does not ensure that variables occurring in reflective fragments are correctly bound in the available environment.

Because such issues cannot easily be left aside in statically-typed languages, these are less amenable to staged metaprogramming. The evaluation semantics upon which their type systems depend are defined in terms of a given processor. Conceiving a type system that fits an evaluation paradigm with an infinite stack of processors evaluating each other took the best part of a decade. Taha's MetaML [86, 88] is a statically-typed, functional, multi-staged, hygienic language. Contrary to previous systems which use implicit operators, it is based on four explicit constructs to control staging:

1. build a *representation* of the code of an expression;
2. *splice* a fragment of code into another;
3. *evaluate* a fragment of code;
4. *lift* an evaluated value as code.

The soundness guarantees it offers are very strong: the multi-staged program is type checked once and for all to ensure safety in all subsequent phases. While MetaML remains the model of staged metaprogramming, other systems have modified some of its properties for other uses. For example, Template Haskell [81] retains MetaML's type safety, but restricts metaprogramming to happen only statically for reasons of performance and represents code as an algebraic datatype.

While not originally intended for domain-specific programming, MetaML-like languages proved to be quite apt at it. See for ex-

ample Czarnecki's comparison of domain-specific programming in MetaOCaml, Template Haskell—two variants of MetaML—and C++ templates [29]. In fact, Ganz et al. showed that MetaML can be seen as a generalised macro system, by defining their MacroML system in terms of MetaML constructs [37]. Similarly, Kim et al. extended ML with LISP-like staged metaprogramming constructs, which they claim is more user friendly, while maintaining strong soundness guarantees [46]. A variation particularly relevant to our concern is Converse [90], a language that mixes staging with a specialised macro system to provide for domain-specific syntax. This language is thereby well suited for implementing domain-specific semantics using staged metaprogramming, and domain-specific syntax using its macro system.

MetaML and its variants are functional languages. Extending safety of a similar nature to object-oriented multi-staged languages is not trivial. Pašalić and Linger proposed a calculus of that nature [72], demonstrating the constrained nature of the resulting object model. Fähndrich et al. describe compile-time reflection for C# [35], a mainstream object-oriented language. It maintains strong soundness guarantees amongst other by requiring explicit control of scopes in the metaprogramming code. Despite that, the expressiveness of metaprogramming is limited, as described in the article's "future work" section.

Other languages inspired for staging by MetaML have compromised its strong safety guarantees in favour of practicality or compatibility. Nemerle [84] is a functional staged language, that is compiled for the `.NET` platform. Because it allows inspecting and modifying declarations directly, its static type guarantees are not as strong as those of MetaML. It does however provide access to the compiler's type checker, making type checking an integral and explicit part of metaprogramming code.

While MetaML-like languages stage the definition of their semantics, Shields et al. [82] extend the paradigm in what they call "staged type inference". There, the type inference of a staged expression is itself staged, so that it happens on the expression during the stage when it is evaluated. The system maintains overall soundness by requiring the inferred type to be compatible with its static approximation. This allows to write staged metaprograms which structure depends on dynamic values—for example dynamically loaded code—, something that MetaML-like languages cannot. Template Haskell, which was already mentioned, is also inspired by this technique. However, the

complexity of staged type inference calls its applicability to mainstream object-oriented languages into question.

Staged type inference results from research on simpler "dynamic typing" for statically typed languages. This was originally introduced by Cardelli in the Amber language [14], and refined by Abadi et al. [1, 2] as well as Leroy and Mauny [61]. The underlying idea is to use a `dynamic` runtime operation that, for a given expression, returns its value alongside a representation of its static type. The program can then define its behaviour in terms of static types.

This form of on-demand dynamic types allows the program to observe the shape of data in a manner that isn't usually available to statically-typed languages. Lammel and Peyton Jones, in their "scrap your boilerplate" series of papers [52, 53, 54] propose a number of operators based on Haskell dynamic types that allow to solve certain problems using a metaprogramming-like approach. These problems—for example data serialisation—traditionally require a large amount of boilerplate code to treat all possible cases. Boilerplate is scrapped by using dynamic types to configure the code to each case at runtime.

The benefit for embedded domain-specific programming of a technique that configures code based on types has already been recognised. Pang et al. use it to implement a domain-specific language for dynamically loading software components in Haskell [71].

4.2 *Related work:* towards a separation of concerns

This section discusses another aspect of metaprogramming: how to design the data structures representing program data in languages that provide direct access to it. This data can either be dynamic, as in reflection libraries, or static, as in Nemerle. In both cases, however, these structures play a particularly fundamental role because they represent the very nature of the language. Language constructs are carefully tailored to suit programmers' needs. Their representation ought to benefit from the same care.

In 2003, Lorenz and Vlissides [62] propose a design for a reflection library that decouples its interface from its implementation to make it pluggable. Simultaneously, Bracha and Ungar [11] provide a series of guidelines for the design of a reflection library:

- its interface should *encapsulate* its implementation, so that metaprogramming code does not depend on it;

- it should *stratify* the programming and metaprogramming systems, so that the latter can be implemented as a separate system;

- there should be *ontological correspondence* between base- and meta-level constructs.

Most reflection frameworks developed before did not follow some or all of these guidelines. Amongst other things, this prevented languages with multiple sources of reflection, despite it having been envisioned in the seminal paper on LISP reflection [85]. Compliance with these guidelines has become standard practice, for example in Coppel's prototype of a Scala reflection library [23].

As was already observed in Bracha and Ungar's seminal article, mirrors provide a solid intellectual foundation to refine reflection methodologies. For example, Ungar discusses how mirror principles were applied in a virtual machine and development environment for Self to improve code reuse and integration [91]. A similar intuition will be at the basis of Chapter 7, which aims to unify some aspect of compilation and reflection.

Finally, and to tie the discussion on reflection with that on staging, some considerations on reflection libraries design also apply to data structures representing code. Denker et al. describe the design of a reflection library that gives access to the bodies of methods [30]. It provides a framework to modify program trees, and a pluggable type system to verify certain metaprogramming properties. Although originating from a very different background, these systems provide similar services to those provided by Nemerle.

I believe that this similarly exemplifies a reality of current metaprogramming research. There exist a variety of solutions to specific metaprogramming problems, from a variety of sources and traditions. We have discussed some in this chapter, and more in Chapter 1. However, while they all aim to solve related issues, larger, underlying trends that may unify them do not yet emerge. A unified theory of metaprogramming would provide tremendous benefits. Staged programming as proposed in MetaML has been a major contribution in that regard. However, as we discussed in this section, it is unclear whether it can function for languages built around mainstream concepts—object-orientation, use of reflection, et cetera. While very limited in scope and pretension, the second part of this dissertation aims to explore some alleys that may contribute to unifying certain aspects of metaprogramming. Our focus on domain-specific programming, which oftentimes combine dynamic and static aspects, will lead us to consider metaprogramming at the border between the compiler and runtime.

Chapter 5

Compiler-Supported Code Lifting

Staged metaprogramming like Taha's MetaML shows attractive properties when it comes to controlling the semantics of a language. Semantics becomes part of the program—it becomes a metaprogram—, so that it can be controlled and defined to suit domain-specific needs. The language's static type system guarantees once and for all that all programs resulting from the metaprogram are correct. Furthermore, if staged evaluation can partially happen during compilation, runtime performance may be very high. However, we are interested in considering MetaML-like abstractions in light of domain-specific programming in Scala and related languages. This raises certain issues:

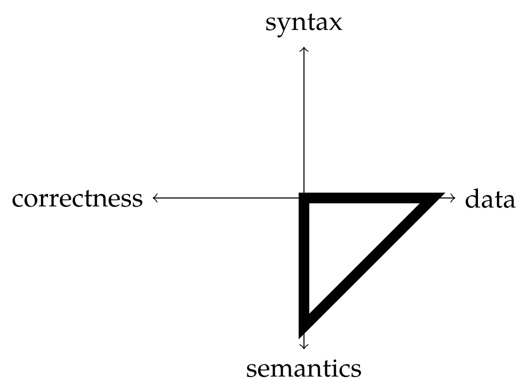
1. staging is explicit, which means that domains cannot be transparently embedded;
2. proposed staging methods that are compatible with mainstream object models are either not type-safe, or restrict metaprogramming possibilities;
3. partial evaluation is difficult to provide in Scala like in other object-oriented languages [80, 79], which reduces prospective performance benefits.

Because of that, MetaML-like abstractions are not a present solution to domain-specific programming in Scala, and may never be. The first two items are particularly worrisome, as they jeopardise our goal of transparent embeddings and the possibility to replicating MetaML's strong soundness properties in Scala.

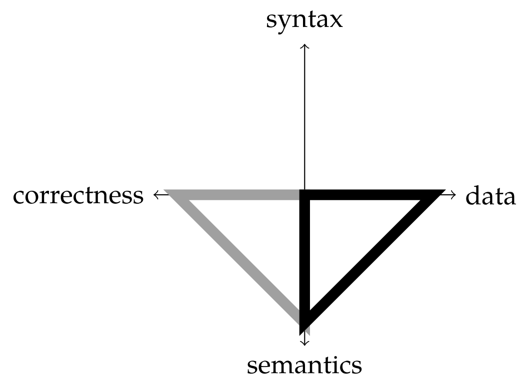
Nevertheless, the staging approach described in this chapter is inspired by Taha's work. But instead of explicit syntactic operators

for staging, it uses the host language's type system to control lifting. This is the main novelty of this approach. On the other hand, it does not propose a solution to implementing full soundness guarantees on the staged metaprogram. Instead, later-stage metaprogramming operations are applied to typed data structures representing the code—lifted code—, thereby allowing further type checking to be done explicitly if needed. This approach is similar to that of Nemerle. Scala's staging does provide support to maintain hygiene—by staging only closures, which automatically lift all necessary references from the environment. As we will see, the technique restricts the number of stages to only two: static and dynamic. This makes staging significantly simpler and, I argue, is sufficient for domain-specific embeddings. There, only two stages are relevant: that of compiling the host language, and that of compiling domain-specific expressions. Like in the examples of Part I, the first stage happens statically, the second dynamically.

Scala's code lifting technique is used for domain-specific languages that utilise the host language's grammar and type system, but change its semantics. To go back to the model of domain-specific embeddings in Chapter 1, it provides for the following embedding coverage.



Note however that, contrary to library-based embedding strategies described in Part I, this technique is not exclusive of others. In other words, it is for example possible to utilise code lifting in conjunction to structural types, thereby obtaining a larger embedding surface.



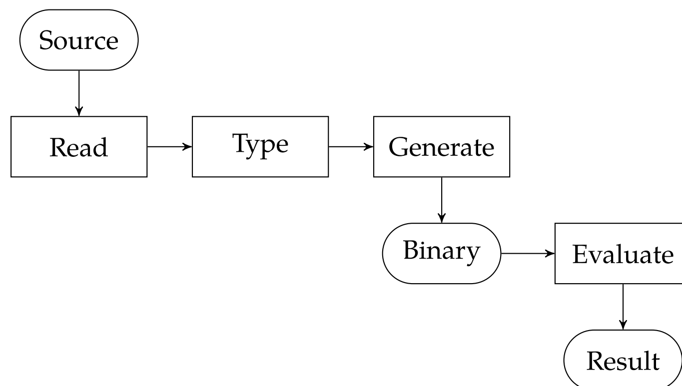
The simple underlying idea of Scala lifting is that expressions of the host program that have a special `Code` type follow a compilation path that lifts them. For lifted expressions, a representation of the code is generated instead of executable code. The `Code` type takes an argument that defines the type of the lifted expression, which is checked to be correct. At runtime, this representation is interpreted by a domain-specific library that implements the required semantics, in a way similar to that described in Chapter 2. For example, this may be done by rewriting the expression to another language and sending it to a specialised tool for interpretation. Alternatively, it may be directly evaluated by an interpreter provided by the domain-specific library.

This chapter will start with a description of the compiler mechanisms needed to implement two-stage code lifting in the Scala compiler. It will continue with a discussion of a usage example that transforms **`for`** comprehensions on XML documents into XQuery expressions.

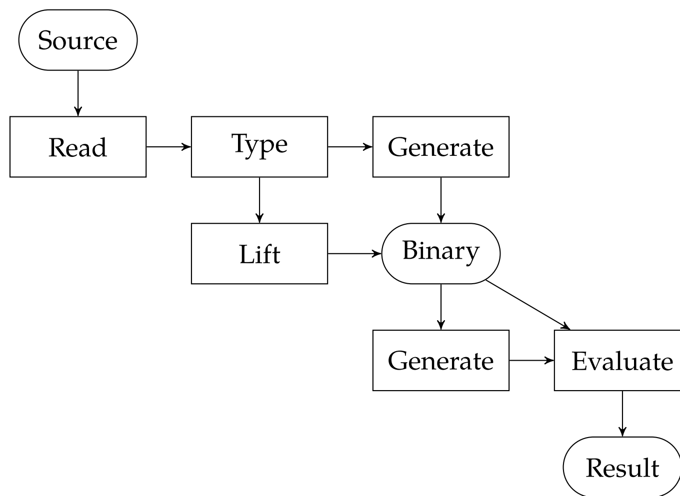
5.1 Staged compilation

To start, let us consider a simplified view of the compilation and execution of a non-staged program. First, the source code of the program is read and transformed into an internal compiler representation, usually an abstract syntax tree. In statically-typed languages, the program is further analysed by the type checker to understand its structure, assign symbols to names and give types to elements of the tree. Furthermore, analysing the program allows to detect incoherences in the structure of the program such as dereferencing undeclared variables or incorrectly calling methods. These first two phases are the front end of a compiler. In the back end, a number of transformations are applied to the program, progressively encoding abstraction using more basic structures. At the end of this process, a binary program is generated, which is devoid of programming language abstractions.

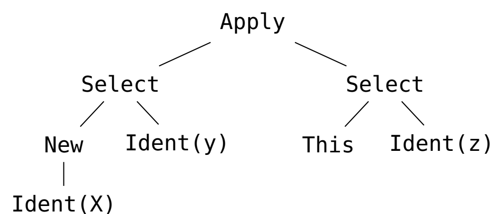
The binary program is evaluated by the machine—either an actual processor or a virtual machine—and a result is obtained.



In a two-stage compilation, the traditional separation between compilation and evaluation is relaxed. Besides machine instructions, binary files contain fragments of abstract syntax trees with symbols and types. Parts of the program can be evaluated directly, but other fragments must first be compiled in what is called a "second stage" of compilation. In principle, the same process can be repeated in the second stage compiler, leading to multiple staged compilation, similar to MetaML. In this chapter however, we will focus on two-stage compilation. As we will see, this simplified form is sufficient for solving many problems related to domain-specific programming. Typically, the first stage of compilation is concerned with general-purpose sections of the program, while the second stage handles domain-specific compilation. The complexity of the programming model for two stage compilation is higher than that of traditional compilation, but remains more manageable than with an arbitrary number of stages.



In a staged program, the compiler front end remains identical to that of a normal compiler. At the end of the type analysis phase, the program is separated into fragments that are generated normally, and others that are lifted to the second stage. The means through which lifted fragments can be differentiated is described later in this section. Lifting then transforms second stage fragments into a binary representation of the compiler's abstract syntax tree. In Scala, this is done by replacing lifted fragments by code that, when evaluated, results in a data structure similar to the abstract syntax tree. Below is a simplified example of this transformation. The graphic corresponds to the abstract syntax tree obtained for the expression `(new X).y(this.z)`



How lifted trees are used during evaluation is very dependent on their meaning in the program. If they represent fragments of domain-specific code, they will usually be processed by the domain-specific library. To do so, it may interpret the trees using domain-specific semantics that differ from those of the host language. Alternatively, it may compile the trees into second-stage binaries that can be evaluated by a specialised, domain-specific machine. This second case is described in the next section, using queries on XML documents as the domain, and an XQuery processor as the domain-specific machine.

In code lifting, the same expression may either be compiled normally or lifted, depending on its role in the program. By themselves, lifted and non lifted expressions look identical. It is their environment that defines their role, so that the program must contain additional information about which expressions are lifted, and which are not.

In MetaML, lifted code fragments are marked syntactically using the bracket operator. For example, `3+4` is evaluated to 7, while `<3+4>` is lifted to the next stage. In Scala, an expression can also explicitly be lifted by calling the `Code.lift` method. It takes an expression of an arbitrary type `T` and returns the lifted expression, of type `Code[T]`. However, Scala's explicit staging infrastructure does not have the richness of MetaML. Only one of its four staging operators is available: that building a representation of code. Splicing or lifting values have not been implemented, although it would be possible. As for MetaML evaluation operator, which controls how code is evaluated over multiple stages, it is unnecessary in a simpler two stage metaprogram.

However, while explicit staging operators are available, the novel method to lift code in Scala is type-driven. Simply stating that an expression has type `Code[T]` will stage it. For example, in the simplified example below, the body of `x` is compiled directly, while that of `y` is lifted.

```
val x: Int = 3*a+b
val y: Code[Int] = 3*a+b
def f(c: Code[Int]) = ...
f(3*a+b)
```

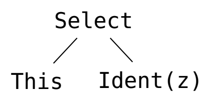
The same technique can be used for method parameters, so that when method `f` is called, its parameter is passed as code instead of being passed as value.

When compared with syntactic code lifting, the responsibility for deciding to lift code moves from the call site to the site of declaration. This is a particularly attractive property for domain-specific programming because the domain-specific library can request lifted code without any involvement of the user. However, we will see in the example below that controlling lifting through types can lead to situations where insufficient code is lifted.

We will now shortly discuss the changes that were required in the Scala compiler in order to implement a prototype of code lifting. The implementation is split into two parts.

1. A compiler phase that replaces calls to the `Code.lift` method by the data structure that represents the lifted expression. In the Scala compiler, this phase is called `LiftCode`.
2. Small modifications to the type checker in order to automatically insert calls to `Code.lift` when appropriate.

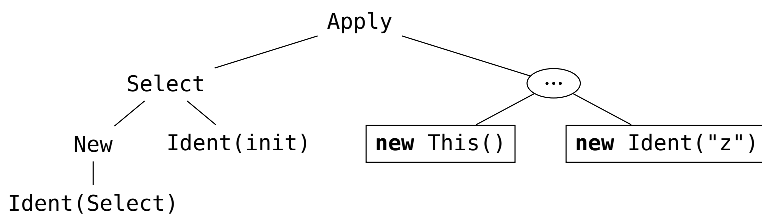
Lifting code The `LiftCode` phase is implemented as a transforming traverser, which searches the entire abstract syntax tree of a program for any call to `Code.lift`. When such an occurrence is found, the subtree corresponding to the method's argument is passed to a reification method. Reification transform an abstract syntax tree and the corresponding symbols and types into a new tree that, when compiled and evaluated returns a representation of the original tree. For example, let us consider that the expression lifted by `Code.lift` is **`this`**`.z`. In the compiler, this expression is represented by the following abstract syntax tree.



Reification will transform this tree into another tree corresponding to the following expression.

```
new Select(new This(), new Ident("z"))
```

This expression, when executed, builds the original abstract syntax tree for **`this`**`.z`. Internally, the reified expression is itself represented as a tree depicted below. The full tree being too large and complex to be of interest, only its left side is fully displayed.



This transformation obviously implies a significant increase in code size. However, its implementation in the compiler's `LiftCode` phase is relatively straightforward:

- the *reifier* rewrites all compiler symbols, types and trees to equivalent `scala.reflect` instances, which are part of the public standard library;

- the *injector* serialises the `scala.reflect` instances as an abstract syntax tree of class constructors and literal values, following the model described above.

Type-directed lifting As was mentioned above, the preferred way of defining staging expressions in Scala is implicit, by giving or inferring a `Code` type. Because `LiftCode` relies on explicit `Code.lift` invocations, these must be added in an earlier phase.

The type `Code[T]` is special, as it is assigned to an expression which doesn't itself return a value of type `Code`, but of type `T`. Let us assume that the Scala type system contains a judgement like the following—there is no formal type system for Scala so that it is illustrative only.

$$\Gamma \vdash e : T \rightsquigarrow e'$$

This judgement gives a type T to an expression e in an environment Γ, while simultaneously rewriting e to e'. The rewritten expression is assumed to have the same type as e. Type-directed code lifting is defined by the addition of the following rule to the judgement above.

$$\frac{\Gamma \vdash e : (\overline{T_p}) \Rightarrow T_r}{\Gamma \vdash e : \texttt{Code}[(\overline{T_p}) \Rightarrow T_r] \rightsquigarrow \texttt{Code.lift}(e)}$$

This rule implements the following properties.

- Only function types ($* \Rightarrow *$) can be lifted—the reason for that is hygienic and will be detailed below.
- The type parameter of `Code` defines the type of the lifted expression, which must be valid within Scala's normal type system.
- An expression of type `Code` is rewritten as a call to `Code.lift`.

Note that the expression resulting from the rewrite is correct by Scala's original type judgement, because of the signature of `Code.lift`:

```
object Code {
  def lift[A](expression: A): Code[A] = ...
}
```

Because the addition of explicit lifting calls is directed by types, possibly including inferred types, this transformation and the type checker are interdependent.

We will not delve into the details of the implementation of Scala's type checker. Suffice to say that the `Code` rule is implemented as an extension of Scala's `Typers` class. Because it is only relevant for expressions with a function type, it is part of the `typedFunction` method. There, it tests for `Code` types to modify typing and activate the addition of `Code.lift`.

On hygiene As was mentioned in the introduction of this chapter, the staging system described here does not aim to maintain a level of type safety like MetaML. It does, however, aim to support hygiene. A staging or macro system is hygienic if definitions in staged fragments or macros are guaranteed not to collide with other definitions upon expansion. Consider for example the following program.

```
val code = {
  val a = 40
  Code.lift({() => a + 2})
}
{
  val a = 0
  evaluate(code)
}
```

If staging is hygienic, the evaluation of `code` will yield 42, if it is not, the result will be two. In the former case, the reference to `a` remains with the value defined in the first block, where it belongs. In the latter case, the reference is rebound to whatever local value of the same name is available, which is most likely not the expected behaviour.

The staging mechanism discussed here is intended to provide a domain-specific embedding library with a representation of code to be evaluated at runtime. The implementation's `evaluate` is domain-specific, but is likely to use a form of dynamic reflection to access values referenced from the lifted fragment. A naive implementation would evaluate the example above by reflectively searching for a local value named `a`, leading it to a non-hygienic implementation. However, the semantics of closures in Scala will have correctly bound the values composing their environment, making them available by reflection to the domain-specific library. Requiring code implicitly lifted through its type to be a closure does not guarantee hygiene but makes it easy and straightforward—the domain-specific library would have to go out of its way to be unhygienic.

5.2 The XQuery for Scala domain-specific language

To give a concrete example of how code lifting can be used as a metaprogramming tool for domain-specific programming, we will discuss a library that implements query shipping for XQuery in Scala. It was implemented by Fatemeh Borran as part of her master's thesis [8], which I supervised. Query shipping is a method through which XML-related code fragments in Scala are transformed into XQuery code and sent to a standalone XQuery database system. This

can greatly improve performance of XML-related operations as the overhead of converting Scala fragments into XQuery code is quickly compensated by the performance of the XQuery engine when dealing with complex queries. Furthermore, because XQuery queries are generated from Scala code, the user of the library needn't learn a new language. This method is a typical example of embedded domain-specific programming where the domain-specific language keeps the host language syntax, but changes its performance semantics and data.

The basic implementation of the query shipping library is relatively straightforward. Scala code that accesses XML data stored on an XQuery database obtains the data using an API-provided `load` call that reads the database. The data can then be treated as a simple set of XML elements, and queried using standard Scala constructs, particularly **`for`** comprehensions. Obviously, performance would be very poor if the data was actually queried by downloading the entire database and then running queries on the data locally. This is why the queries must be rewritten. To do so, the API requests lifted code for queries by declaring parameters with `Code` types, when needed. The Scala compiler automatically lifts the query expressions. At runtime, the abstract syntax tree corresponding to the Scala code is built as described in the previous section. Internally, the library contains a compiler that transforms lifted Scala trees into XQuery query strings. Finally, these can be sent to the XQuery system, returning the same value that would have been calculated by the Scala expression without lifting.

To give an example of the use of this library, consider the program below.

```
Transform.trans{ () =>
  for (
    val b <- load("bib.xml") \ "book"
    b \\ "price" < 10.0
  ) yield b
}
```

Method `trans` is the library's lifting and XQuery compilation method. The library does not directly execute XQuery queries but instead generates Scala abstract syntax trees, which, when evaluated, run the XQuery query string on the database system. This implementation therefore assumes the presence of a Scala interpreter, which was not part of Borran's system. This being said, the resulting value of the program above is the abstract syntax tree corresponding to the code that follows.

```
load(new java.io.StringReader(XQuery.run(""""
  for $b in doc("bib.xml")//book where
  $b/price <  10.0
  return $b
""""")))
```

The transformation from Scala `for` comprehensions to XQuery code is done through a number of rewrite rules described in detail in Borran's thesis. We will not discuss the tree transformation logic here, as it is of little interest to the matter at hand. However, the remainder of this section will consider some weaknesses of the implementation that are relevant to code lifting for domain-specific programming in general.

Because XQuery is a limited language, not all Scala programs can be transformed into XQuery. In such situations, the library generates an error at runtime. In principle, this issue may be circumvented by using a best effort approach to domain-specific transformation. The largest fragment of the query code that has a corresponding XQuery expression is transformed, the remaining part is left in Scala. Obviously, this process requires that the library transforms a syntax tree to another tree, as is done in this library, and not directly to XQuery strings.

Another limitation is that the transformation covers at most one lifted expression. Oftentimes, it would be more efficient to generate a single database query that answers multiple program queries. This allows the optimiser of the database system to run more efficiently, and may reduce the amount of data being transferred between the server and the application. The lifted code approach described above allows such an implementation. By not transforming lifted code to XQuery eagerly, the transformation may happen on larger code fragments obtained by merging previously lifted queries. It should be noted, however, that creating useful larger queries that can then efficiently be optimised by the database is non trivial, and beyond the scope of Borran's thesis or of this discussion.

Borran's library also suffers from that it does not explicitly consider the problem of free variables in lifted code. By assuming the existence of a syntax tree interpreter, it simply generates trees that reference free variables. For example, consider the following query, which is similar to that above except for the fact that the lifted expression contains a free variable `x`.

```
val x = 100.0
Transform.trans{ () =>
  for (
    val b <- load("bib.xml") \ "book"
    b \\ "price" < x
  ) yield b
}
```

The trans method will return for the lifted expression the following code.

```
load(new java.io.StringReader(GalaxTest.run(""""
  for $b in doc("bib.xml")//book where
  $b/price < """" + x.toString +
  " return $b"
)))
```

This expression simply references the identifier x to copy its value into the XQuery query string. The code must then be interpreted within the context of the original expression. To do so, the required interpreter would have to obtain the value of x through reflection or a similar mechanism.

Finally, the library also demonstrates one issue with type directed code lifting. Method trans is defined in a way similar to the following.

```
def trans[T](code: Code[T]): Code[T] = ...
```

This means that whatever expression is passed to trans will automatically be lifted. However, this property does not automatically propagate further than the argument of the method. For example, in the following code, the lifted code fragment will simply be “q”.

```
val q = { () =>
  for (
    val b <- load("bib.xml") \ "book"
    b \\ "price" < x
  ) yield b
}
Transform.trans(q)
```

The relevant code fragment will not have been lifted by the compiler, and will therefore be unavailable. To correctly compile the example above requires the user to annotate value q with type Code[NodeSeq]. We see in this example that type directed code lifting, as described above, is not in practice transparent to the user.

5.3 Discussion and analysis

Metaprogramming on the structure of code brings the ability to embed domain-specific programming that control the semantics of the host language. This form of metaprogramming is popular because it solves domain-specific problems without programmers needing to learn specialised domain-specific languages. The host language's syntax remains the same, its semantics becomes domain-specific. Of course, this requires that the changed semantics remain meaningful with the original syntax. In practice, as in the example of the previous section, the domain-specific semantics are often indistinguishable from that of the host language, and only modify compilation.

Microsoft's LINQ system [9] gives an idea of such an ability's attraction. LINQ is part of the `.NET` 3.5 framework, where it compiles host language expressions—for example in C#—to SQL [51] or, like Borran's system, to XQuery [16]. To do this, it uses a series of technologies, a key one being the ability to extract expression trees. It was released about one and a half year after Borran's work, and proved to be immensely popular because it hides the complexity of the platform. Although an application may access data from a remote database, code can be written as if that data was stored in the program's own memory and using its own data structures. In Scala like in LINQ, using types to mark lifted fragments makes domain-specific programming mostly transparent to the user.

While sufficient for LINQ-like problems, the type-directed code lifting technique is limited in its ability. When compared to MetaML, it cannot offer the same soundness guarantees or an unbounded number of metaprogramming stages. For library-based domain-specific programming, these limitations are acceptable, because two stage programming is sufficient to implement such libraries, and because its dynamic nature is unlikely to allow for strong type safety guarantees anyway. However, there are at least two limitations that are problematic for domain-specific programming. Both have been pointed out in the example of the previous section.

1. Although closures favour hygiene, domain-specific libraries are still required to use reflection to reference declarations from lifted closures. The lifting framework does not provide direct support to access these values.

   A possible future extension is to utilise the knowledge at compile time, during reification, to adorn reference in the abstract syntax tree with getters when appropriate. These getters are defined using runtime reflection, which relies on the names of the

closure class and of its lifted variables. This information is easily accessible at compile time.

2. Only expressions whose indicated or inferred type is `Code` are lifted, but no code that depends on them. If the lifted expression is, for example, a reference to a method, lifting does not provide a way to obtain the code of that method. If the body of the method ought to be staged, the user will have to mark it as `Code`, thereby exposing the staging. This is similar to what happens with MetaML, but is undesirable for domain-specific programming.

   A possible solution would be to store an abstract syntax tree representation of all methods and fields. References in lifted code fragment could either be treated as opaque, or further queried for their code. Storing all trees would be costly in terms of binary size, but may not be unacceptable since code can be stored in very compact form [48].

The solutions to both problems share the property that they expand the interface of lifted code to allow further exploration dynamically. To better support domain-specific programming, code lifting brings static information to the runtime of the program; a theme that has been thoroughly discussed in the previous chapter. However, we see here that this static information cannot live in isolation once it enters the dynamic environment. It needs to tie into the environment. How lifted code and other metaprogramming constructs can be tied into a coherent system will be the subject of Chapter 7.

This chapter describes a metaprogramming solution to access code. However, similar embedding characteristics can be obtained with a more lightweight metaprogramming approach. Rompf et al. have described language virtualisation [75, 15], which is such a solution in Scala. There, domain-specific expressions are built within a virtualising trait, which redefines certain types and operations to lift syntax in a manner similar to ZyTyG (Chapter 2). However, instead of reducing the host's syntax to tokens and parsing them, language virtualisation retains the structure of the code as a dynamically-built abstract syntax tree. The virtualisation trait is an abstract interface for the domain-specific semantics. The expression's actual domain-specific behaviour is controlled by mixing the trait with others that define rewrites or evaluation strategies. Scala's mix-in types allow for a modular and fine grained control of what rewritings or evaluation are applied, and in what order. This allows the semantics of virtualised

code to be controlled in a manner not unlike that afforded by multi-staged metaprogramming. On the other hand, language virtualisation suffers, like ZyTyG, from its inability to control certain aspects of the language which semantics are hard-coded, particularly those related to side effects and control flow. Recent work by Rompf [73] and others has led to making non-virtualisable operators more friendly to this technique. This is obtained by modifying the compilation strategy for these operators to resemble that of Scala's `for` loop. Instead of fixed semantics, their compilation is defined, at least conceptually, in terms of a series of method calls, with the library providing default implementation for the methods—and therefore default semantics for the operators. This allows domain-specific expressions to virtualise these operators by defining the required implementation methods.

Language virtualisation has the advantage of being a pure library approach to embedding domain-specific programming, which does not require compiler-supported metaprogramming. However, to virtualise the whole language requires specialised abstractions that are not particularly relevant to general-purpose programming. Furthermore, while code lifting may provide typed abstract syntax trees directly, virtualisation requires types to be added subsequently. This can be a problem for certain domain-specific problems. But we will discuss in the next chapter another form of metaprogramming that concentrates on providing type information.

Chapter 6

Static Types At Runtime

In Part I and in the previous chapter, we have seen various reasons why static type information is desirable to access at runtime.

- Access type information about the environment when dynamically type checking embedded domain-specific fragments.

- Execute domain-specific code that relies on host language types to define part of its semantics, such as with structural types representing a relational algebra.

- Obtain types to attach to trees when lifting code using language virtualisation.

If domain-specific programming is to be provided through libraries, a mechanism to access the static types of the program will be required. The solution that this chapter considers is inspired by dynamic types in Haskell and ML, such as those used for the "scrap your boilerplate" technique. These ideas where previously discussed in 4.1. In this section, we will see how they can be applied in a mainstream object-oriented language.

Traditional reflection gives access to the types of programming interfaces. For example, in Scala, one can reflectively list the members of a given class, their type, the super type of the class: all type information that the compiler knows about the class. However, reflection does not give access to types of expressions, which are not part of interfaces. This prevents runtime access to detailed type information held by the compiler. Particularly, it prevents obtaining instantiations of abstract types—type parameters of classes or methods, or abstract types. For example, consider the simple example below.

```
object A {
  def f[T](x: T): T = x
}
A.f(new B)
```

Through reflection, it is possible to obtain the signature of `f` in terms of the type variable `T`. When `f` is called with a parameter of type `B`, the type variable `T` is instantiated, and `f` gains a concrete type for the duration of that call. The concrete type for `f` is statically available at the call site. However, because a method call is an expression, and is not part of the interface that can be reflected, this static information is not available at runtime. Only a simpler erased type is present, which merely approximates the original type, as we had seen in Chapter 3. In particular, this prevents a runtime library to obtain a concrete type for a call to `f`. In the example above, this concrete type would have been `(B)B`—the type of a method similar to the function `B => B`. At runtime, all abstract types have necessarily been concretised, but this basic information cannot be accessed.

Because of that, using traditional reflection to obtain static type information for domain-specific programming has the following drawbacks:

1. the library cannot gain static type information about domain-specific expression themselves, since the types of expressions are not maintained for the reflection library;

2. the library cannot obtain the concrete type of a declaration that a domain-specific expression refers to, if it is defined in terms of type variables or abstract types.

In practice, this makes type information from traditional reflection all but useless for domain-specific embedding libraries. Another form of metaprogramming is needed.

Runtime types A possible solution is to make all internal compilation data available at runtime. This is what happens in virtual machines with runtime types, that is, where every value is adorned with a representation of its full type like that used by the compiler.

However, Scala, like many strongly typed languages, is compiled using type erasure. In this process, the original types of the program are replaced by simpler ones encoding a type discipline that is compatible with, albeit less precise than, the original type discipline. For example, when Scala code is compiled to Java virtual machine binaries, Scala specific types such as mixins are replaced by Java classes

and interfaces that the virtual machine understands. Compiling a program by erasing its types has many practical advantages.

- A rich type discipline suitable for programming may not be optimal for evaluation; erasing it into simpler types that are particularly suited for evaluation can improve performance.

- Also, separating the type system of the language from that of the evaluation machine allows more flexibility in language design. The Java virtual machine, which supports a simple class based type discipline, is the host for a multitude of languages with extremely varied type systems.

But these advantages are counterbalanced by the fact that a reflection library for an erased language—or other operators that expose the types of the program—will not have access to the static structure of the program. Indeed, much of the relevant information has been lost in translation as erasure is not a bijective transformation that would allow original types to be reconstructed.

A possible solution is to modify compilation to generate alongside the program a representations of types in the form of program data. This differs from a typed virtual machine in that the representation of types has no specific meaning for the evaluation machine. Doing this allows to use program types to define elements of the semantics of the language, without making the machine dependent on a given type discipline. In Scala, such a system was proposed by Schinz in his thesis [78]. However, the performance penalty of this approach is considerable. Schinz reports slowdowns on representative programs ranging from 15% to 86%. Important increased in code size (21–49%) and memory allocation (33–360%) are also reported. The underlying problem comes from the fact that library provided run time types are an “all or nothing” solution. In practice, most code can do without runtime program types as is demonstrated by current programming practices in erased languages.

We discussed in the previous chapter the idea of staged programming. There, fragments of code are lifted on demand to be available at runtime. In this chapter, we will discuss a system built on the same premises, but to access typing information rather than code. Representation of types as runtime data structures are generated only when needed, removing the performance penalty of classical runtime types.

6.1 Type manifests

A type manifest is a data structure representing static type information at runtime. They are created by lifting program types during compilation, where they are available. This section describes manifests, and the method that controls their generation. Manifest are available in version 2.8 of Scala, albeit in a relatively simple form. This chapter describes both the existing Scala manifests, and an extended implementation thereof.

The Scala standard library contains a `Manifest[T]` data type. An instance is a run time representation of type `T`. A very basic trait representing manifests is below.

```
trait Manifest[T] {
  def erasure: Class[_]
}
```

A manifest represents a complete Scala type, which obviously requires a more complex data structure. For now however, let us observe that to integrate with the Java reflection library, a manifest provides the erasure of the full Scala type it represents. Even though the erasure of the manifest is a Java reflection `Class` instance, it provides information that is not available using Java reflection. We will start by discussing this difference as it is key to understanding the particular nature of manifests in metaprogramming.

The dynamic type of an instance can be read through Java reflection. But the types used in the program itself differ from the runtime types. More precisely, the type of a runtime instance at a given location in the program is a subclass of the static program type for that location. For the purpose of type checking a domain-specific fragment, it is the program type of the environment that is of interest. Consider for example the example below, using the embedded SQL language.

```
def f(a: Any) =
  SELECT ('name, 'firstname) FROM 'people WHERE 'age < a
f(if (state) 4 else (new Cow))
```

To check the domain-specific expression requires to validate the fragment `'age < a`, which depends on the host program parameter `a`. Type checking the example above based on the static type correctly rejects the SQL expression—cows should not be involved in this matter. However, if using the runtime type, the program may or may not be rejected, depending on the value of `state`. Worse, if the value changes

between checking the expression and evaluating it, the expression may be checked as valid and still fail at evaluation.

In general, using traditional Java reflection, it is not possible to obtain the static type of `a`. On the other hand, manifests can provide the correct information. The example below shows how this can be done and introduces the discussion on automatically generated manifests.

```
def f[T: Manifest](a: T) =
  SELECT ('name, 'firstname) FROM 'people WHERE 'age < a
f(if (state) 4 else (new Cow))
```

First, notice that the program has been changed to contain more static type information. Instead of weakening the static type of argument `a` to `Any`, a type parameter `T` is introduced to represent the compiler's knowledge about the type of the argument at the call site of the method. While this allows the static type information to be propagated further in type-checking, it is not sufficient to make it available at runtime. It is by bounding the type parameter `T` with the context `Manifest` that a manifest representing the static type of `T` can be obtained. Calling `manifest[T]` within the body of `f` will provide the manifest corresponding to the current method call. The table below lists a variety of calls to `f` and the corresponding manifest available in the body of `f`.

Call	Manifest
`f(4)`	`Manifest[Int]`
`f(List(1, 2, 3))`	`Manifest[List[Int]]`
`f(if (state) (new A) else (new B))`	`Manifest[Object]`
`f(new A with B)`	`Manifest[A with B]`

The manifest does not simply reproduce information from the source; it represents a fragment of the model of the program that has been built by the compiler. The example above containing a conditional expression makes this property very explicit: type `Object` reported in the manifest is the least upper bound of **`new`** `A` and **`new`** `B`, a type that the compiler calculates as part of the model of the program but isn't explicitly provided by the developer. In code that uses only basic types such as early Java code, the type information added to the model is small when compared with that of the source. However, in code that heavily relies on type parameters or other advanced type system features, the amount of information contained in the program model can be considerable. For domain-specific code, it is critical to have access to this information because domain-specific libraries are

generally implemented as abstract libraries and greatly benefit from obtaining concrete type instantiations relative to their use.

The manifest for argument `T` is bound to the type that the argument is instantiated with at the call site. This demonstrates the staged nature of manifests. On the one hand, the type represented by the manifest is static. On the other hand, its flow through the program follows the rules of its evaluation. In fact, the manifest can be seen to move to the next stage at the point where the manifest is instantiated—the method call in the example above.

Context bounds In the example above, we assumed that in the presence of a `Manifest` context bound, a manifest is available. To understand the reason for this, let us take a step back to understand context bounds. Context bounds are part of the implicit programming infrastructure of Scala. As first approximation, they can be understood as syntactic sugar whereby the following structure:

```
def f[T: Manifest](...) = ...
```

is transformed into the following code:

```
def f[T](...)(implicit mt: Manifest[T]) = ...
```

In other words, a context bound defines an additional, implicit parameter of the type of the context bound applied to the type of the variable it bounds. By making the manifest parameter implicit, it need not be provided by the programmer at the call site. Instead, the compiler will automatically complete the call by passing a value as argument to the implicit parameter, provided the value demonstrates the following properties:

1. it is itself marked as implicit;
2. it is in scope and accessible at the point of use;
3. it is of the correct type;
4. it is the only possible value—if not, the implicit parameter is ambiguous.

In other words, once a corresponding manifest value is passed to a method call bound by a manifest, it will then automatically be woven into the fabric of the program through subsequent bounded method calls, without additional programmer intervention.

While using context bounds brings us closer to our goal, it still requires the programmer to provide the original manifest value for the implicit parameter to be passed. This is why the manifest infrastructure contains another element to have the manifest values be generated when needed by the compiler. If the implicit argument cannot be completed according to the rules described above, the

compiler will create a fresh implicit value based on its own knowledge about the type. This allows the static type information calculated by the compiler to be introduced into the program flow. As an example of this, let us consider the example below, where `f` is the manifest-extracting method that we already used above.

```
val ab = new A with B
def g[U: Manifest](u: U) = f(u)
g(ab)
```

The compiler will rewrite this code as follows.

```
val ab = new A with B
def g[U](u: U)(implicit mu: Manifest[U]) = f(u)(mu)
implicit val mab =
  Manifest.intersectionType (
    Manifest.classType(classOf[A]),
    Manifest.classType(classOf[B])
  )
g(ab)(mab)
```

There, `mab` is the synthetic value created by the compiler to hold the manifest for the type of `ab`.

The manifest does not need to be instantiated by the compiler inside the body of `g` because the manifest `mab` is implicitly available from the previous call and can be used further. This is good because, if a manifest had to be created for the body of `g`, it would be for the type variable `U` and not an actual instantiated type. While correct from the point of view of typing, it is not useful to have a type manifest representing a type variable. As in the example above, type variables must be forwarded from the location where they are statically instantiated to the point where they are needed by using method arguments. By designing manifests at the boundary between static and dynamic evaluation, their ability to provide a useful type is increased.

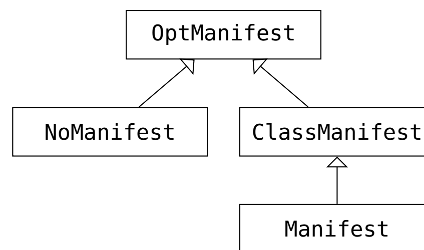
Controlling manifest generation Useful manifests cannot always be generated. Consider for example the code fragment below.

```
def f[T: Manifest](t: T) = ...
def g[U](u: U) = {
  f(u)
}
```

At the call site of `f` in the body of `g`, the only information available about the concrete type for `T` is that it is equal to the type variable `U`. If `U` was itself context bound with a manifest, Scala's implicit

inference mechanism would usefully forward it to `f`, according to the logic described above. But since a dynamically passed manifest is not available, a manifest must be generated using static information: the manifest passed to `f` encodes an abstract type. Because manifests are to be used to concretise types, this is generally not the expected result. The user ought to annotate `U` so that its concrete value is available at runtime. In that case, the compiler should fail because it is missing a manifest.

However, there are some cases where manifests are desirable but not necessary, or where manifests representing abstract types are actually useful. As an example, manifests may be used to optimise domain-specific code based on static information, but a less optimal implementation is available in the absence of that information. This is for example the case in the implementation of Scala's invariant arrays, which utilise manifests to map compatible arrays with native Java virtual machine arrays. To cater for these various use cases, manifests in Scala come in different flavours.



The `Manifest` class can represent all Scala types[1], including abstract ones. If a `Manifest` is requested and it is not implicitly available, the compiler will always generate one based on its static type information. In the example above, this would be a manifest representing the type variable `U`.

A `Manifest` is a generalisation of the `ClassManifest` type. This latter type is able to represents a subset of all Scala types: those which can be instantiated on the Java virtual machine. In practice, this comprises all class types, including instantiated parametric classes—for example `List[Int]`, class `List` instantiated with the parameter `Int`. It excludes abstract types, as well as existential or intersection types.

Because not all types can be represented by a `ClassManifest`, the compiler cannot always create a fresh manifest from static type information when the implicit does not resolve. This is why an `OptManifest` context bound ought to be used when a `ClassManifest` is required. If the manifest is available implicitly, or if the compiler can

[1]The Scala 2.8 implementation does not support structural types

generate it, a `ClassManifest` is provided. Otherwise, the absence of a type that can be instantiated is represented by the `NoManifest` object.

```
def f[T: OptManifest](t: T) = ...
def g[U](u: U) = {
  f(u)
  f(new A)
}
```

The example above demonstrates these two situations. The manifests generated for `T` in the body of `g` are listed in the table below.

Call	Manifest
`f(u)`	`NoManifest`
`f(new A)`	`ClassManifest[A]`

Compiling manifests To support manifests requires changing the compiler to generate code for creating manifests when needed. The change plugs into Scala's implicit parameters inference mechanism, itself part of the `Typers` phase, the primary type-checking phase of the compiler. Implicit search is a subtle process whose detailed behaviour is beyond the scope of this discussion. It suffices to say that the `bestImplicit` method is used to generate a tree fragment referencing the implicit value whose use is most likely to yield a correct program. However, in the case of implicit manifests, two situations can arise in `bestImplicit`:

1. it has found an implicit value of the correct manifest type, and will return a reference to it;

2. no implicit manifest value of the correct type is available in scope.

In the second case, the expression returned by `bestImplicit` to satisfy the implicit parameter is a factory for a new manifest instance. The correct constructor is selected by the `manifestOfType` method.

The standard library contains, for each of the manifest kinds, a set of factory methods for all types that the manifest can represent. The following object contains some of the factories for class manifests.

```
object ClassManifest {
  def classType[T](clazz: Class[_]):        ClassManifest[T]
  def classType[T](clazz: Class[_],
                   args:  OptManifest[_]*): ClassManifest[T]
  ...
}
```

For the compiler, generating an expression to call the first version of `classType` is straightforward. The `clazz` parameter is a literal value representing the erased type of the class. However, the second version of `classType` demonstrates the recursive nature of most manifests. There, the manifests that represent the arguments to the class may themselves be implicitly available, as in the example below.

```
def f[T: Manifest](t: T) = ...
def g[U: Manifest](u: U) = {
  f(List(u))
}
```

In the body of `g`, a manifest representing type `List[U]` must be generated. The `args` argument of the second `classType` method is the manifest that is passed to `g` to represent its parameter `U`. This is obtained through another recursive call to `bestImplicit`. In other words, the recursive nature of types is replicated in the recursive nature of implicit resolution for manifests.

6.2 Revisiting type-safe relational algebra

As an example of using manifests, we shall revisit the type-safe relational algebra domain-specific language of §3.4. We recall that the library allowed to write relational algebra queries and execute them in a type safe manner, as in the example below.

```
def studentsInLesson =
  project[{
    def firstName: String
    def lessonName
  }] { person join attends }
}
for (p <- query(studentsInLesson)) {
  println(p.firstName + " " + lessonName)
}
```

However, without using metaprogramming, the `query` operator could not be implemented. Indeed, for every row returned by running the query on the database, it must create an object that is compatible with the result type of `studentsInLesson`. This type is the following structural type.

```
{
  def firstName: String
  def lessonName
}
```

For reasons explained alongside the original example, it is conceptually possible to create an instance of that type, because the domain of a relational algebra gives it a meaningful semantics. However, it requires the library to access the static type of `studentsInLesson` at runtime. Using manifests in conjunction with reflection solves this problem.

To do so, the domain-specific library requests a manifest for the type of the relation. The signature of method `query` is as follows.

```
def query[T: Manifest](r: Relation[T]): Set[T]
```

With this information, the implementation of the method, albeit non-trivial, can be done using traditional reflection techniques. The implementation below is at a very high level of abstraction, but demonstrates the general idea underlying it.

```
def query[T: Manifest](r: Relation[T]): Set[T] = {
  val resultSet = db.execute(r.toSqlString)
  val resultType = manifest[T]
  for (r <- resultSet) yield {
    newProxy(resultType) { (name, args) =>
      r.getField(name)
    }
  }
}
```

In this example, the query defining the relation is executed on the database `db` using an equivalent SQL expression. This mechanism is explained in detail in Lavanchy's report [59]. A `resultSet`—which is an untyped data structure—is returned by the database. It may be a JDBC `ResultSet`, or a similar structure. The result type is then obtained from the manifest representing the type of `T`, that is, the type of a single row in the resulting relation. At this point, data and types are both available, but separate. The final **`for`** loop merges them using proxies. A proxy is a special type of class provided by the Java virtual machine, which implements a given interface, and responds to method calls using an explicitly defined dispatch method. Here, the `newProxy` method hides the implementation of the proxy. It uses a manifest to define its type. This assumes that a Java interface can be generated for every Scala type, which is true but may require runtime code generation. Such proxies ought to be part of any reflection library, but their implementation is beyond the scope of this chapter. The second argument of `newProxy` is an anonymous method that implements method dispatch. To do so, it uses the name of the called method to access the correct field in the `resultSet`. Because

the domain-specific library guarantees type safety for the relational algebra, this access always succeed.

6.3 The manifest manifesto

The example of the previous section shows how manifests contribute to implementing domain-specific embedding libraries. It also demonstrates how, once static type information has been made available at runtime, specifically compiler-related problems appear in this phase too. This is exemplified by the use of a proxy in the implementation of the relational algebra library. In a statically-typed language, instantiation is normally controlled by the compiler, which generated the declaration of the type as a class, and maintains a reference to its symbol to generate instantiation code. With proxies, this whole process is moved to runtime.

Does this mean that libraries and metaprogramming transform a statically-typed language so thoroughly as to have it behave as if it where dynamically typed? No, on the contrary: manifests, code lifting or structural types are methods that maintain the statically-typed nature of both the host language and embedded domain-specific languages. From a user's perspective, the type-safe relational algebra library provides an exceptionally thorough level of static soundness guarantees. However, to provide this integration requires the domain-specific embedding library to take charge of some of the compiler's tasks.

Manifests are a crucial tool to allow such hybrid designs. It is by providing domain-specific libraries with static type information that such libraries can implement type-safe domain-specific languages. In a dynamically-typed language, type information is easily accessible to the runtime—there is no separation between static and dynamic information. Manifests do not break down the separation between compilation and evaluation. In any compiled language, information about the static structure of the program implicitly is part of the dynamic structure. Manifests make this ordinarily implicit link, explicit. In that, they do not change the nature of statically-typed language. Information flows in the same manner in the system; for example, domain-specific libraries cannot change compilation using manifests.

Though the distinction between compiler and runtime remains, the nature of the compiler changes. Instead of being a black box whose only connection to the world is binary code, manifests—as well as lifted code—open cracks that allow metaprograms to peak into its

internal structure. The manifests we have discussed in this chapter are very minimal: they provide the erasure of their type, as well as some details on the structure of complex types, as we discussed concerning parameterised class instances. Manifests ought to provide more information. For example, a manifest representing a structural type contains a list of members that define the structure of the type. These members are themselves annotated with types, that refer to classes, and so on. For obvious reasons, it is not possible to store all information related to a manifest in that manifest. Instead, a manifest must contain just enough information to reconstruct other related data. This data—definitions, types, values—is present in interfaces that are part of the program's normal binary representation, unlike the data carried by manifests. Reflection can be used to access it, if the manifest allows to plug into the reflection library. It is therefore crucial that manifests be integrated in a reflection framework. Moreover, they are used to implement processes similar to those of a compiler. The integrated framework ought to provide services to comprehend and modify structures representing code or types, like those of the compiler.

The next chapter discusses how manifests, code lifting, reflection, and other metaprogramming tools can be brought together into a coherent system. There, manifests are not merely packets of information left by the compiler for domain-specific libraries. Instead, they serve when needed as pegs that clamp the dynamic model to the static one, where otherwise it would have been approximated.

Chapter 7

A Unified Program Manipulation Framework

In previous chapters, we have seen two forms of metaprogramming particularly relevant to embedding domain-specific languages: code lifting and manifests. We discussed how these abstractions provide additional metaprogramming information beyond that available through general reflection. Reflection provides access to information about the runtime state of the program, and its static structure as visible through interfaces. Code lifting and manifests provide access to detailed compiler data about code and types, which had statically been requested by the metaprogram.

At this point in the discussion, our metaprogramming abstractions have been conceived separately from each other and from general reflection. The resulting set of tools is disparate, creating a programming paradigm that exposes much implementation complexity to the user. To restate an example from the previous chapter, a manifest—representing a type—does not correspond to a type in the language's reflection framework. Similarly, a reference to a variable in lifted code does not provide a method to read its value using the reflection framework. The user is made responsible for implementing the relation between the various metaprogramming tools.

This chapter considers a design for a metaprogramming framework that unifies manifests, code lifting and general reflection. In previous chapters, I described manifests and code lifting as small fragments of compiler data left in the code for runtime libraries. This points to the fact that a unification of these techniques with general reflection is akin to unifying portions of the compiler's data structures with those of reflection. A static symbol table and a dynamic reflection interface are two data structures that represent the same fundamental concept: programs. It is therefore natural

that the same abstract interfaces be used for both, although their implementations may differ. This provides the desired unification at the level of public interfaces. Tools and services defined at that level—such as class readers, algorithms on types, tree utilities, et cetera—can easily be shared between the compiler and the various parts of the metaprogramming framework. However, as we will see, the implementation required to support a strong notion of identity for the unified interface is non-trivial. Both reflection and compiler symbol tables are dependent on a strong notion of identity.

The next section discusses the various elements that may compose a unified metaprogramming framework. It explores their similarities, and proposes a model based on Bracha and Ungar's mirror-based reflection to unify them and exploit commonalities. It proceeds with a discussion on the complexity that stems from using multiple sources—static and dynamic data—to create a unified representation that requires absolute internal consistency. In section §7.2, I propose a possible implementation of the unified design, based on the Scala compiler and data structures similar to virtual classes. To conclude this chapter, we will reconsider the resulting system in light of domain-specific programming and of the metaprogramming model it offers.

7.1 Metaprogramming with mirrors

To start, let us consider the design of the components of a typical compiler and of a typical reflection library that are of interest to this discussion. As in previous chapters, the concrete discussion will refer to Scala, albeit, in principle, it could serve as blueprint for any similar, statically-typed, erased language. Because of the unification task we are attempting, this does not only comprise the language, but also the actual implementation of its compiler. Of particular interest here is the process through which the compiler obtains a symbol table and abstract syntax trees from source, which was outlined in §5.1. The discussion on the general reflection library will pertain to standard Java reflection when needed, or to a prototype of mirror-based reflection for Scala, which is directly mapped upon concepts from the language specification [33].

As was discussed above, a runtime metaprogramming framework must combine at least the following elements into a coherent system.

1. Manifests, which are precise, non-erased types for specific elements of the program. Their representation is derived from the compiler's static representation. Furthermore, if code rewrite

is needed by the domain, abstract syntax trees lifted from the compiler must also be part of the framework.

2. The interface of the program, which is pickled (stored) in the program's binary files. This representation is similar to, but not identical, to the symbol table of the compiler. Furthermore, the compiler has a component, the unpickler, which reads pickled interfaces into a symbol table.

3. Information about the runtime state of the program. This includes existing instances, values of fields, the call stack, etc. In the Java virtual machine, this information is available through the Java reflection library. The model of the program that is assumed by the virtual machine is not that of the symbol table, but the result of its erasure.

4. Beyond reading data about the program, a complete metaprogramming framework must act on the structure of the program. This is provided by the Java reflection library, for changing values, instantiating objects and calling methods. More complex changes which add classes or change their structure, require code generation, for example using the compiler.

The first two forms of data are obtained through different means. However, they share their fundamental structure and logic with the symbol table of the compiler. Furthermore, core components such as the unpickler, are identical in the compiler and metaprogramming framework. Until now, these systems have been implemented with very little code reuse. The representation of types in manifests is a custom implementation that is unrelated to the compiler's representation. Existing metaprogramming frameworks in Scala have reimplemented their own unpickler, their implementation being usually very specific. The representation of programs in existing reflection libraries, such as that of Java or of prototypes in Scala are unrelated to those used by the Java or Scala compilers. This section and the next consider how to unify the static component of reflection with manifests—and lifted trees when applicable.

The design I propose to unify manifests and interface reflection can be summarised as follows.

1. The standard library of a language contains an abstract interface that represents program structures. This includes types, symbols, possibly trees, and all related structures.

2. Common tools that act upon program structures, such as the unpickler, are implemented in terms of the abstract interface, and are part of the standard library.

3. The compiler and the reflection library create concrete implementations of the interface tailored to their own use.

As we will see, implementations in themselves can be very lightweight, especially if the implementation language supports virtual classes. The 2.8 version of the Scala compiler partially follows these principles. Core data structures such as symbols, types and trees are implemented as abstract interfaces in the `scala.reflect` package. The class file unpickler only depends on this interface. The compiler's symbol table completes the `scala.reflect` package with compiler-specific features. Demonstration code for a reflective implementation of `scala.reflect` also exists, but is not complete enough to support metaprograms.

A mirror-based design The paradigm proposed by Bracha and Ungar for mirror-based reflection §4.2 is a foundation for my work. The usefulness of the paradigm as a means for unification had already been recognised by Ungar et al. [91], who uses it to unify reflection, debugging and other services. As we will see, encapsulation, stratification and ontological correspondence, the three properties that underlie mirrors, come together to support unified metaprogramming.

1. An abstract metaprogramming interface and concrete implementations is the very definition of an encapsulated design. In that sense, the nature of a unified metaprogramming framework coincides with a key properties of mirror based reflection. The rationale put forward by the proponents of mirrors for that property is that it allows to change the reflection provider, for example to replace a local implementation with a remote one. Unified metaprogramming pushes that idea further by generalising it to parts of the compiler itself.

2. In traditional, non-mirror based reflection libraries, reflection mechanisms are weaved into the runtime structure of the program. For example, in Java's reflection library, access to the `Class` of an object—a key reflection provider—is done through a normal instance method of `Object`. This would not be possible to reproduce in the abstract metaprogramming interface. Indeed, it assumes that the reflected program is running, which is not true in the implementation of the interface by the compiler.

Therefore, a stratified implementation is not only suitable, but needed.

3. A key concern with a unified metaprogramming design is for the abstract interface to support equally well the needs of the compiler and reflection. While not actually solving the issue, basing the abstract interface directly upon the structure of the language is a first step in making unification possible. The concept of "ontological correspondence" advocated by mirror-based reflection serves the same purpose, albeit for reasons of usability rather than to support unification. It must be noted that this is a problem in the current Scala compiler as its symbol table does not completely follow the structure of the language as defined in the language specification. For example, refinement types and mixin types are two different concepts in Scala's language specification. However, they are implemented as a single datatype called `RefinedType` in the compiler.

These three points show that the principles that underlie mirror-based reflection generalise extremely well to a unified metaprogramming framework.

On identity However, there remains another key design principle that needs to be taken into account: Both the compiler and the reflection library must handle coherent sets of program elements. This concept is oftentimes implicit in simple systems, but more complex ones must make it explicit. Interestingly, Bracha and Ungar do not speak of this issue in their paper.

In the Scala compiler, this notion is defined in terms of compilation "runs", whereby the symbol table takes a coherent state after having compiled a closed set of files and their dependencies. Usually, the compiler only survives for a single run, but some uses such as IDE support or interpreters require that it isn't restarted entirely for each run. Multiple batches of files are submitted to the same compiler, which is then said to be in resident mode. A new compilation run on the same files will not yield the same symbol table. It may be structurally equal, but its identity may differ. Indeed, the symbol table heavily relies on instance identity, and therefore requires that only one instance of each symbol exists. Interestingly, early versions of the Scala compiler did not clearly differentiate compiler runs, which was the source of many a problem.

In a reflection library, such as that of Java, a similar problem of internal consistency is visible. More precisely, the library demonstrates

a problem that already arises in the virtual machine: classes need to be unique. If two `List` classes are loaded by the virtual machine, they will define different types and their instances will be fundamentally different. In the Java virtual machine, and its reflection library, this issue is handled by class loaders. Each class loader hierarchy contains a coherent set of classes. However, the same class loaded in two loaders will be treated as distinct, for example preventing an instance of `List` from the first class loader to be passed to a method that expects a `List` from the second.

To abstract runs and class loaders, we must build upon the framework of mirrors. In Bracha and Ungar's proposal, each element—class, object, method—is mirrored by itself. To pursue the metaphor, this situation is dangerous because, if each element is reflected on its own, separate mirror, the program will be visible from different angles, which may yield an incoherent picture. Instead, it is more correct to consider a single mirror that reflects all elements: the perspective remains identical for all reflection and no incoherence can arise. In this metaphor, a mirror corresponds to a class loader, or to a compiler run. The elements it reflects are coherent amongst themselves, but not with those from other mirrors. To prevent confusion, I will call a mirror that abstracts class loaders and compiler runs a "universe".

In this model, the difficulty stems from maintaining a coherent universe even when it is created from multiple sources. If a manifest refers to a class, and the same class is read using reflection, the resulting class mirror must be the same instance. Only one instance of each symbol is created; it is however acceptable to have multiple instances for the same syntax tree or type because they do not carry a notion of identity like symbols. Of course, it is in principle possible to build a representation of a program that does not follow these rules, and where symbol equality is recalculated every time. However, such a design is inherently inefficient, making it unpractical to use in a compiler. Because the abstract metaprogramming interface is to be shared by the compiler and reflection library, and because the former requires efficient data structures, out design must provide represent symbols as unique instances.

To better grasp the situation let us consider a pseudo-Scala example that checks if a manifests represents a single class and, if so, returns the result type of the method called `f`.

```
manifest[T] match {
  case ctp: ClassManifest =>
    ctp.classSymbol.findMethod("f").resultType
  case _ => Nil
}
```

The type represented by the manifest is generated within the universe of the compiler. The type that is obtained by calling `resultType` on a method is generated within the reflective universe unpickled from the classes' interface. However, despite their different origins, these two universe are one in a unified metaprogramming framework. Let us consider what happens in this code if the instantiation of type `T` is the following class.

```
class A { def f: A = ... }
```

Here, the result type of `f` is the type of the class itself. Both types refer to the symbol of `A`, which must be the same instance despite their different origins. A unified metaprogramming framework must therefore abstract over the implementation in such a way that identity can be maintained. Assuming that each provider can guarantee identity for its own data, it can be maintained overall if each symbol is assigned a unique origin.

In our model, there is an abstract interface shared between the compiler and metaprogramming framework. However, because we consider compiled languages, we can assume that the two do not share metaprogramming data. This means that, if a symbol mirror for class `A` is created in the compiler and is lifted to runtime in a manifest, it need not be the actual instance that is lifted. Instead, the compiler generates code that creates a new instances within the metaprogramming universe at runtime. Symbol identity must be maintained between manifests, lifted code, and data from general reflection, but not across the compilation boundary. In a compiled language, even in presence of metaprogramming, there remains a barrier between static and dynamic evaluation which guarantees that they cannot be interleaved.

In a Java-like object-oriented language, any method or field is necessarily a member of a class, and is therefore part of its interface. This always allows general reflection to be used to create a mirror for it. Maintaining identity for such symbol is therefore a question of delegating the task of creating them to a single origin. A manifest or lifted code fragment that creates a mirror for a symbol should not instantiate it directly. Instead, it should be defined in terms of a call to the corresponding factory method in the general reflection

framework. Because both systems are defined as part of a unified universe, symbol mirrors from that framework will be compatible with the the data structure of the manifest or lifted code.

There remains a problem with local variable symbols, which may not be part of the interface of the program. However, as we had seen in the paragraph on hygiene of §5.1, a reference in a fragment of lifted code to a free variable is transformed into a field of the closure encapsulating the lifted code. This symbol is therefore accessible though the general reflection framework. Only the symbols of references to variables that are defined within the lifted fragment cannot be instantiated in that manner. Since the declaration and all uses of the symbol are, by construction, part of the lifted fragment, it is possible to assign the origin of that symbol to the lifted code itself. Because any fragment can only be lifted once, the uniqueness of the symbol is guaranteed.

In the case of the metaprogramming infrastructure described in this dissertation, a unique origin can be assigned for each symbol a priori. Symbols of methods or fields originate within the general reflection framework. Symbols of local variables bound within lifted code fragments are generated as part of the fragment. If the metaprogramming infrastructure were to be extended to include more sources of data, designers must ensure that symbols can still be assigned to an origin a priori.

7.2 An implementation to share code

The previous section described a mirror-based design for a metaprogramming framework, that unifies the compiler and various parts of the reflection framework. The claimed main benefits of that design are twofold:

1. metaprogramming code can be written in terms of the abstract data structure, and shared between implementations;

2. multiple implementations can coexist alongside each other.

In this section, we will discuss in more detail how the principles laid out in the previous section can be implemented. The first element of the implementation is to use an encoding of the abstract interface inspired by virtual-classes. The second, is to use the unified abstract interface as a foundation to connect multiple metaprogramming layers, and thereby unify multiple implementations. Before we discuss these two elements, let us quickly consider the interface of the abstract data structure for metaprogramming.

Data structures for the abstract interface In a design for a metaprogramming framework, the abstract metaprogramming interface is crucial. It defines the interface for representing program features that is shared by the compiler and the public reflection library. This work bases its abstract interface on the existing model of the symbol table in the compiler. This is a pragmatic approach, since this data structure cannot be amended easily. We will however reconsider this design in the conclusion of this chapter, and again in the conclusion of this thesis.

The structure representing programs in the compiler is composed of three major categories of data: symbols, types and abstract syntax trees. Besides these three major categories, it also uses additional types such as the following:

- names that underlie symbols but are not unique;
- scopes that contain member symbols within class symbols or structural types;
- flags that control various properties of symbols;
- constants that are used to represent literal values in trees or for constant types;
- annotations that may be applied to trees or symbols.

Each of the major and additional categories are defined either as a single class or trait, or as a trait hierarchy. All traits composing a type are defined within a module-trait, and their dependencies are defined using a self-type annotation on the module For example, a skeletal implementation of the module for types may contain the following code.

```
trait Types { self: Universe =>
  abstract class Type { ... }
  class ClassType(...) extends Type { ... }
  class CompoundType(...) extends Type { ... }
  ...
}
```

A coherent, single universe is composed by mixing all traits into a class, in a manner similar to the example below.

```
class Universe extends Symbols
                  with Types
                  with Trees
                  with ...
```

This design implements Scala's *sandwich pattern*, which is outlined in Chapter 27 of the Scala guide by Odersky et al. [69]. This pattern also exists in other language, for example in C++ [43].

An encoding like virtual classes The design of the abstract interface must be such that it allows writing generic code. This means that code is defined purely in terms of the interface, independently of its implementation. A simple set of abstract classes is sufficient to write generic code that only uses method calls. But useful generic code must also be able to instantiate objects, or to deconstruct them using pattern matching. When generic code is applied to a concrete metaprogramming framework—for example to the compiler or to a reflection library—the instances it creates must be of the corresponding type in the concrete universe. This behaviour is akin to virtual classes [63]. Scala does not natively support virtual classes, but there exist encodings displaying similar properties. Below, we will discuss such an encoding, which uses abstract types as well as factory and extractor methods to provide instantiation as well as pattern matching. This encoding has the advantage of making concrete implementations very straightforward. It is used in the Scala prototype implementation for types, but other categories of data are implemented differently.

An abstract version of this data structure cannot be obtained using simple object-oriented extension. Take for example the module for types described above, and assume it is defined abstractly. Can a concrete instance of the abstract symbol table be created by extending it? Trait `Types` can of course be extended with a concrete implementation, alongside each abstract class that compose it. However, this design does not provide the desired level of abstraction. Let us for example consider the following two simple universes: `AbsUniverse` is the abstract interface to metaprogramming, `ReflectedUniverse` is a concrete instance of it. For the sake of this example, only types and symbols are defined.

```
abstract class AbsUniverse extends AbsTypes with AbsSymbols
class ReflectedUniverse extends AbsUniverse
                             with ReflectedTypes
                             with ReflectedSymbols
```

A metaprogramming tool—for example a class file unpickler—ought to be written in terms of the `AbsUniverse`, as in the example below.

```
abstract class Unpickler {
  val universe: AbsUniverse
  def unpickle(...) = {
    ...
    universe.ClassType(...)
    ...
  }
}
```

Here, the body of the `unpickle` method creates, amongst other things, a class type by calling a factory method in the abstract universe. This implementation is independent of the eventual universe, which is defined by giving a value to `universe` when `Tools` is made concrete.

```
object UnpicklerForReflection extends Unpickler {
  val universe = new ReflectedUniverse
}
```

However, for this code to work requires the abstract universe to be implemented as more than a set of abstract classes.

The basic idea of the design is to utilise abstract type members to represent data structures, and to utilise Scala's support for object construction and deconstruction in `apply` and `unapply` methods. To understand how this work, let us consider for example the module that defines the abstract interface for types—to simplify, the hierarchy of types has been reduced to only two.

```
trait AbsTypes { self: AbsUniverse =>

  type Type >: Null <: AnyRef

  type ClassType >: Null <: Type
  val ClassType: ClassTypeExtractor
  abstract class ClassTypeExtractor {
    def apply(s: Symbol): ClassType
    def unapply(t: ClassType): Option[Symbol]
  }

}
```

The trait `AbsTypes`, and its self-type are identical to the concrete implementation above. However, instead of using classes to represent a `Type` or `ClassType`, abstract type members are used. This says that a class implementing `AbsTypes` will need to provide concrete types for `Type` and `ClassType`. In the implementation we refer to, `Type` was an

abstract class and does not need a factory, but `ClassType` does. This is provided by the `ClassType` value—companion of the type because it has the same name—, which must implement the `ClassTypeExtractor` interface. The extractor defines an `apply` and `unapply` method, which have a special meaning in Scala as demonstrated below.

```
val t = ClassType(s)
t match {
  case ClassType(s) => ...
  case _ => ...
}
```

In this example, the expression `ClassType(s)` serves as constructor for the type and will be rewritten as `ClassType.apply(s)`. The case in the pattern match expression is rewritten as `ClassType.unapply(t)`; the case matches if the unapply returns some value, otherwise, the default case is executed. It is thereby possible for the abstract universe to define constructors and de-constructors, and implement generic tools like the unpickler above.

A concrete implementation of module `AbsTypes` for reflection is obtained as follows.

```
trait ReflectedTypes extends AbsTypes {
  self: ReflectedUniverse =>

  trait Type extends TypeImpl

  case class ClassType(s: Symbol) extends Type
  object ClassType extends ClassTypeExtractor

}
```

Abstract type members of the abstract interface are directly implemented as traits or classes. Companion values are implemented as objects, whose constructors and de-constructors are automatically implemented by the **`case`** modifier on the corresponding class. Concrete instances of the abstract interface can thereby be implemented using remarkably short code. Of course, if a similar framework were to be implemented in a language supporting virtual classes, this encoding would become unnecessary.

Multiple layers The design described above gives the ability to easily create many concrete implementations of an abstract metaprogramming interface. It fulfils the goal of allowing to write generic metaprogramming code, which is independent of its implementation,

in accordance to the principles of mirror-based design. However, a universe can only be instantiated to a single implementation.

```
val universe: AbsUniverse = new ReflectedUniverse
```

Here, the universe is based on Java reflection and class signature unpickling. But the second design goal for the unified metaprogramming framework is to be able to unify multiple sources of metaprogramming into a single one.

```
val universe: AbsUniverse = new UnifiedUniverse {
  val reflection = new ReflectedUniverse
  val lifting = new LiftedUniverse
  ...
}
```

The example above shows a concrete instantiation of a universe that provides metaprogramming using both a reflection-based universe and one handling code lifting. Symbols and types would be provided by reflection, while trees may come from code lifting. Both `ReflectedUniverse` and `LiftedUniverse` are implementations of `AbsUniverse`. But `UnifiedUniverse` is also an implementation of the same abstract interface. The unifying nature of a mirror-based design means that merging universes becomes a simple question of implementing a forwarding proxy. While generally straightforward, two issues remain to be considered.

1. How to support data structures in one universe which contain references to values from different universes.

2. To which sub-universe should the unified universe forward requests for creating new instances.

The first issue is solved by making the implementations of universes generic to the universe. Like an unpickler tool that can be written generically for any universe, the code of a concrete universe is written such that it can accept elements of another universe. For example, the `ClassType` type in the previous example contains a symbol. There, the symbol is required to originate from the same universe as the type. This is because the constructor defined using the `apply` method of the companion value takes a symbol that is linked to the current universe.

```
trait AbsTypes { self: AbsUniverse =>
  ...
  abstract class ClassTypeExtractor {
    def apply(s: self.Symbol): self.ClassType
```

```
    ...
```

Although `self` has type `AbsUniverse`, it defines a constraint that forces the instance of the universe owning the symbol to be the same as that owning the type. To allow symbols from other universes requires that the parameter of `apply` is abstract.

```
    def apply(s: AbsSymbol): self.ClassType
    ...
  }
}
```

The `AbsSymbol` type represents a symbol from any universe. It is defined as part of the `AbsSymbols` module.

```
trait AbsSymbols { self: AbsUniverse =>
  type AbsSymbol = u.Symbol forSome { val u: AbsUniverse }
  type Symbol >: Null <: AnyRef
  ...
}
```

In this implementation, I use a Scala existential type that does not put any constraint on the universe owning the symbol. This design allows for the desired architecture, but is weak in terms of typing. Indeed, there is no guarantee that a unified universe will only contain data from one of the sub-universes composing it. Existential types are too generic in that regard. Proposing a more refined design is future work. A direction for that work may be to consider a form of union types that could be applied to the prefix `u` of `Symbol`. In Scala, the prefix can either be exactly one universe or, using existential types, any universe. If a type that defines the prefix as a closed set of acceptable universes were available, it could be used to implement the desired constraints.

The second issue for defining the forwarding proxy that implements a unified universe is to consider instantiation. The discussion on identity in §7.1 laid out the property that, for any element in the universe, either multiple instances are allowed, or it is possible to know a priori which universe is responsible for it. This considerably simplifies the question of forwarding instantiation. Every `apply` method defined in the unified universe may either not care about which universe instantiates the object, or may calculate the universe a priori. The following code implements the module for symbols in a unified universe implementing `AbsUniverse`.

```
trait UnifiedSymbols extends AbsSymbols {
  self: UnifiedUniverse =>
```

```
  abstract class Symbol {
    val universe: AbsUniverse
    val delegate: universe.Symbol
  }

  object Symbol extends SymbolExtractor {
    def apply(): Symbol = new Symbol {
      val universe: reflection.type = reflection
      val delegate = universe.Symbol()
    }
    def unapply(s: self.Symbol): Boolean = true
  }

}
```

As is expected from a forwarding proxy, each symbol contains a delegate. The universe that provides the delegate is also stored, providing the delegate with a concrete type that includes the universe that owns it. The example above assumes that symbols must always be created by the `reflection` universe. A more complex a priori rule is implemented by creating `Symbol` instantiation code for every universe, and choosing the correct one using an arbitrary predicate.

The ability to define abstract universes in a largely generic fashion offers great freedom to create complex webs of metaprogramming data structures from multiple sources. However, unifying universes require somewhat more boilerplate code than straightforward implementations, as is common for any design based on forwarding proxies. Also, as we previously discussed, Scala's type system does not currently permit to enforce that a unified universe only contains data from one its sub-universes.

7.3 Leveraging unification

As was laid out in the seminal paper on mirror-based reflection, the design is a powerful unifier. The abstract interface allows to easily support multiple implementation of metaprogramming: manifests, lifted code fragments and general reflection. The end of the previous section demonstrated how the unifying nature of mirrors favour the integration of multiple metaprogramming concepts.

Mirror-based metaprogramming, as opposed to reflection, is an extension of the design to the compiler's own data structures. It extends the benefits of the original design to a broader issue. First, the design helps to supports metaprogramming abstractions that

create a bridge between the compiler and runtime. Because a single abstract representation is used as basis for similar data structures in the compiler and the reflection library, no ontological mismatch hinders the flow of information. Second, the design reduces code duplication, because metaprogramming code that is shared between a compiler and a reflection library can be written against the abstract interface. In the Scala prototype, the class file unpickler is the main component that has been made generic. But many other tools may be subjected to the same treatment. Portions of the type checker may be made generic, thereby providing the ability to test type properties in metaprogramming.

In fact, it is most of the compiler that may be made generic, opening the possibility of easily generating binary code from within the metaprogramming framework. Conversely, if metaprogramming code is written using the same abstractions as the compiler, static evaluation of metaprogramming code becomes possible. This opens new avenues for deploying pluggable type systems, custom compiler transformations, et cetera on the basis of partial evaluation. This in turn may play a central role in domain-specific programming. We have seen throughout this thesis how statically-typed languages are suitable recipients for domain-specific programming. Their major caveat is that, because some domain-specific evaluation happens during the runtime phase, embedded domain-specific fragments do not demonstrate all properties of a priori correctness and performance that normal code does. In a sense, they take advantage of the static information, without returning the favour. To give domain-specific embedding libraries the opportunity to do so will require that the compiler understands them better. We have discussed in this part of the thesis mechanisms that give more static information to dynamic libraries; maybe the compiler ought to grab some dynamic information back. How to do this is beyond the scope of this dissertation. However, a likely prerequisite is a unified metaprogramming framework spanning the compiler and reflection, like that described in this chapter.

Conclusion

Because of the variety of domains that may be used in software, it is unimaginable that they may be supported by a single, universal technique. In this dissertation, we discussed a series of techniques that function together to allow a relatively broad class of domain-specific languages to be embedded in Scala. Variations of these techniques may be used in other statically-typed, object-oriented languages, if the right abstractions are available.

The first part of this thesis exemplified how libraries can be used to provide embeddings for domain-specific languages. In these techniques, a library of the host language allows to compile code that has the look and feel of a domain-specific language.

The ZyTyG technique uses simple sequences of method calls and values to represent expressions in an arbitrary language. The structure of the embedding library defines the grammar of the domain-specific language, and how domain-specific values are mapped within the host. This technique does not allow to embed any language, but the class of embeddable grammars is surprisingly useful, and can be defined formally as a member of the modulo-constrained family. The biggest drawback of this technique is that it prevents using the host language's type system to guarantee domain-specific correctness properties.

The next technique focuses on that last point, using Scala's new structural types, as well as compound types, to encode a domain. The types not only serve to provide correctness guarantees but become an integral part of the domain-specific language. However, structural types are not natively supported by the Java virtual machine, and an implementation thereupon is described. The description of this reflection-based implementation also raises the question of the relation between library-based embedding techniques and domain-specific programming.

Library-based embeddings are like tricks of a magic show: the substance of the host language does not change, but the attention of the programmer is drawn in such a manner that he sees another lan-

guage. Host language abstractions are not used to clarify the program by providing fixed, recognisable structures. Instead, their purpose in that method is to break down the language's original meaning in order to later reconstruct new structures around a different mental model—that of the embedded domain-specific language.

Albeit powerful, abstractions of general-purpose languages cannot represent all facets of all domains. The thesis identifies a number of recurring difficulties when embedding languages using libraries without metaprogramming.

- The ability to define domain-specific correctness properties in the embedding library is limited because it lacks access to static information about how the domain-specific fragment integrates with the rest of the program.
- Embedding domain-specific semantics requires the library to get leverage on the domain-specific fragments, which is only possible if they have a different syntax. This may not be desired.
- If domain-specific fragments reference external values, methods, binding must happen according to the rules of the host language—using implicit views in ZyTyG—because implementations of domain-specific libraries cannot access values directly. A similar issue arises when instantiating new objects in domain-specific code.

The recurring issues that have been identified can be considered as different instances of a wider problem, whereby insufficient static information about the program is left at runtime for domain-specific libraries to be implemented. To overcome these issues requires that static data on types, codes or environments be left in the program by the compiler. The second part of the thesis discusses two metaprogramming abstractions that carry code and types from the compiler to the runtime.

Code lifting, a pragmatic approach to staged metaprogramming, lifts fragments of code out of the normal compilation and replaces them with a representation thereof. For runtime libraries, lifted fragments are simple literal values in the shape of an abstract syntax tree, which can freely be manipulated. A novelty of the approach described in this thesis is that the selection of lifted fragments is not done using explicit operators, but implicitly through types. This is well suited for domain-specific programming, because embedding libraries can request fragments to be lifted without the user's knowledge.

Manifests are value parameters that represent the type parameters of generic methods, thereby reproducing static information at runtime

which isn't available in erased languages. The compiler automatically completes these parameters using the knowledge it has inferred about the program's types. But because manifests are parameters, they are then passed according to the runtime semantics of the program. The static information left over by the compiler merges with an actual runtime execution to concretise abstract parameters. This gives domain-specific libraries access to information about types that is as precise as that used by the compiler, yet specific to a particular instantiation.

As such, the usefulness of these two techniques is limited because they do not form a coherent programming model. Mirror-based metaprogramming is a proposed design that extends mirror-based reflection to unify it with the compiler. This create a coherent model for domain-specific metaprogramming, encompassing reflection, code lifting and manifests.

This framework must be understood not only as a technical solution to improve manifests and code lifting, but as an attempt to solidify the scheme to embed domain-specific languages using libraries and metaprogramming. The very nature of the design that exposes the internal structure of the compiler through an abstract interface, integrates domain-specific metaprogramming in the heart of a language, where it belongs. A language and compiler that are designed according to this principle include reflection in their design. But the unification is not limited to reflection-based metaprogramming, as library-based tools may also directly be integrated in the framework. For example, an implementation of the abstract metaprogramming interface may be based on lifting code using Rompf's language virtualisation, thereby integrating it with other reflection-based tools for domain-specific programming.

The various tools and techniques of this dissertation form a conceptual scheme to write embedded domain-specific languages using libraries and with the help of metaprogramming. It remains to be seen how successful this scheme will be. The idea of using libraries for embedding domain-specific languages is certainly popular, and ZyTyG, or variations thereof, are commonly used. Code lifting using libraries, as described by Rompf et al. (see §5.3) under the term of "language virtualisation", and which fits the framework very well, has recently been the centre of a major effort to bring more domain-specific languages to Scala. However, metaprogramming techniques for domain-specific programming remain limited in mainstream languages. In Scala, manifests have only been added recently and remain incomplete, and code lifting is available only as a prototype.

The scheme is therefore missing a major component, which hinders its acceptance. Indeed, a main complaint against ZyTyG-like domain-specific languages is the lack of correctness guarantees. As for structural types to represent domain-specific properties, their use without dynamic metaprogramming is limited to providing front-ends without implementations.

To conclude, let us take one step back and consider some lessons that have been learnt through this thesis and that concern the design of languages. My reflection on the importance of domain-specific programming for the future of software lead me to conclude that abstractions that support it ought to be at the centre of general-purpose language's design. They ought to actively support libraries that change their semantics, syntax, the way the environment is accessed, or what properties a program must demonstrate to be deemed correct. As we discussed, MetaML-like metaprogramming and syntax macros remain an attractive and elegant solution. However, it is a radical design which has not been demonstrated to integrate with existing, mainstream programming tools, techniques, libraries, et cetera. The scheme for domain-specific programming described in this dissertation builds on existing, mainstream technologies. It has been described for Scala, a language that is completely compatible with existing Java programs and technologies, making it a good candidate to bring domain-specific programming to the mainstream. What have I learnt about the design of languages that aim to support library-based domain-specific embeddings?

- Elements of syntax are too important to programming to take them away from programmers. Any identifier that is a keyword with fixed semantics is lost to the programmer, and because it was chosen by the designer of the host language, it is likely to be desirable to the domain-specific programmer too. Languages that support domain-specific programming should minimise the number of their keywords, or give the possibility to override their meaning.

- Core abstractions such as calling methods or creating instances should not have a visual character that is too strong, so as to bring flexibility in the way they are read by programmers.

- A static type system is an invaluable tool for domain-specific programming because it allows to cleanly differentiate between the treatment of domain-specific fragments and the rest of the program. A powerful type system allows for finer distinction of

domain-specific fragments and the host program, and thereby tighter embeddings.

- In particular, a language should have conversion abstractions that are guided by types and not explicitly visible in code, such as Scala's implicit conversions. These allow to hide barriers erected because of implementation, but irrelevant to the program, which is crucial to domain-specific programming. In general-purpose programming, this invisibility is oftentimes deemed undesirable because it hides elements of the design. However, library-based embeddings rely on abstraction trickery to draw the attention of programmers to domain-specific properties that may differ from the true structure of the code.

- Types can be used to define domain-specific correctness properties, not only to control the scope of the embedding. There is some indication that extensible types such as structural types, and operators such as union or intersection—Scala's compound types—may be well suited to certain classes of domains. However, it is important that the language provides the ability to create instance of any type, assuming the programmer can provide a semantic for the instance.

- Finally, metaprogramming is a key part of the problem, and should be at the very centre of the design of the language. The data structures for metaprogramming should be designed alongside and in interaction with the syntactic form, and with the same care. A compiler that represents the program in a manner that contradicts the way in which the programmer sees is unsuitable for domain-specific metaprogramming.

It is clear that most mainstream object-oriented languages are not well suited to host domain-specific programming. This appears not to be because this family of languages is inherently incapable of doing so. However, it requires a number of commonly held assumptions about the design of languages to be questioned. For example, statically-typed languages should not only be considered as a mean to guarantee increasingly precise properties on programs, but also as a substrate to drive implicit constructs that disrupt the visible structure of the program. Indeed, there is a need for abstractions that *destructure* programs instead of structuring them, for example in the way syntax is used. With this in mind, the first part of the conjecture underlying the thesis can be fully appreciated.

> Modern statically-typed object-oriented languages such as Scala have language abstractions—or can support new abstractions—that allow to satisfactorily host domain-specific programming. Domain-specific syntax, semantics, correctness properties and data can be provided without preprocessor or custom compiler…

Destructuring the host language permits the implementation of domain-specific embedding libraries, but must be counterbalanced by such libraries restructuring it on the basis of their domain. For domain-specific libraries to reconstruct this structure requires them to function at the same level as a compiler. They must act on types, code and data. This happens in collaboration with the compiler because the fragments they consider do not exist in a void, but within a larger program.

> However, more static data must flow to the runtime implementation of domain-specific libraries than what is required by other libraries. Metaprogramming that unifies the compiler and runtime reflection provides the right framework to support this.

Bibliography

[1] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the 16th symposium on Principles of programming languages*, pages 213–227, 1989.

[2] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5:111–130, November 1995.

[3] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming*, chapter 10–11. Addison-Wesley, 2005.

[4] Giuseppe Attardi, Cinza Bonini, Maria Boscotrecase, Tito Flagella, and Mauro Gaspari. Metalevel programming in CLOS. In *Proceedings of the Third European Conference on Object-Oriented Programming*, pages 243–256, 1989.

[5] Jonathan R. Bachrach and Keith Playford. The Java syntactic extender. In *Proceedings of the 19th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 31–42, 2001.

[6] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proceedings of the 5th International Conference on Software Reuse*, pages 143–153, 1998.

[7] Jon Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[8] Fatemeh Borran-Dejnabadi. Efficient semi-structured queries in Scala using XQuery shipping. Master's thesis, EPFL, 2006.

[9] Don Box and Anders Hejlsberg. *LINQ: .NET Language-Integrated Query*. Microsoft, February 2007. URL `http://msdn.microsoft.com/en-us/library/bb308959.aspx`.

[10] Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of the workshop on Partial evaluation and semantics-based program manipulation*, volume 37 of *ACM SIGPLAN Notices*, pages 31–40, March 2002.

[11] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 331–344, 2004.

[12] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 16th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383, 2004.

[13] Laurent Burgy, Laurent Réveillère, Julia L. Lawall, and Gilles Muller. A language-based approach for improving the robustness of network application protocol implementations. In *26th International Symposium on Reliable Distributed Systems*, 2007.

[14] Luca Cardelli. Amber. In *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 21–70, 1986.

[15] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *Proceedings of the international conference on Object oriented programming systems languages and applications*, pages 835–847, 2010.

[16] Michael Champion. *.NET Language-Integrated Query for XML Data*. Microsoft, February 2007. URL `http://msdn.microsoft.com/en-us/library/bb308960.aspx`.

[17] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 18th symposium on Principles of programming languages*, pages 155–162, 1991.

[18] Edgar Frank Codd. A relational model of data for large shared data bank. *Communications of the ACM*, 13(6):377–387, June 1970.

[19] Charles Consel and Laurent Réveillère. A DSL paradigm for domains of services: A study of communication services.

In *Domain-Specific Program Generation*, number 3016 in Lecture Notes in Computer Science, pages 165–179, 2004.

[20] Steve Cook. Software factories. Slide Show, January 2005.

[21] Steve Cook. Separating concerns with domain specific languages. In David E. Lightfoot and Clemens A. Szyperski, editors, *Modular Programming Languages*, volume 4228 of *Lecture Notes in Computer Science*, pages 1–3. Springer-Verlag, 2006.

[22] Steve Cook, Gareth Jones, Stuart Kent, and Alan Cameron Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007.

[23] Yohann Coppel. Reflecting scala. Semester project report, January 2008.

[24] Jesús Sánchez Cuadrado and Jesús García Molina. Building domain-specific languages for model-driven development. *IEEE Software*, 24(5):48–55, September 2007.

[25] H. Conrad Cunningham. A little language for surveys: constructing an internal DSL in Ruby. In *Proceedings of the 46th ACM Southeast Regional Conference*, pages 282–287, 2008.

[26] Michael A. Cusumano. The software factory: a historical interpretation. *IEEE Software*, 6(2):23–30, March 1989.

[27] Krzysztof Czarnecki. Overview of generative software development. In *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 326–341, 2005.

[28] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, pages 25–39, 1998.

[29] Krzysztof Czarnecki, John O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, template Haskell, and C++. In *Proceedings of the International Seminar on Domain-Specific Program Generation*, pages 51–72, 2004.

[30] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. *Journal of Object Technology*, 6(9), 2007.

[31] Gilles Dubochet. On embedding domain-specific languages with user-friendly syntax. In *Proceedings of the 1st Workshop on Domain-Specific Program Development*, pages 19–22, 2006.

[32] Gilles Dubochet. Computer code as a medium for human communication: Are programming languages improving? In *Proceedings of the 21st Working Conference on the Psychology of Programmers Interest Group*, pages 174–187, 2009.

[33] Gilles Dubochet. Frontend API of Scala reflection library. github.com/dubochet/scala-reflection/frontend, November 2010. URL `https://github.com/dubochet/scala-reflection`.

[34] Burak Emir. scala.xml. Online book, 2009.

[35] Manuel Fähndrich, Michael Carbin, and James R. Larus. Reflective program generation with patterns. In *Proceedings of the 5th international conference on Generative programming and component engineering*, 2006.

[36] Steve Freeman and Nat Pryce. Evolving an embedded domain-specific language in java. In *Companion to the 21st symposium on Object-oriented programming systems, languages, and applications*, 2006.

[37] Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: type-safe, generative, binding macros in macroml. In *Proceedings of the sixthinternational conference on Functional programming*, volume 36 of *ACM SIGPLAN Notices*, pages 74–85, 2001.

[38] Joseph Gil and Itay Maman. Whiteoak: introducing structural typing into java. In *Proceedings of the 23rd OOPSLA conference*, pages 73–90, 2008.

[39] Samuel Z. Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. In *Proceedings of the IEEE*, volume 93, pages 342–357, February, 2005.

[40] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of dsls. In *Proceedings of the 7th international conference on Generative programming and component engineering*, pages 137–147, 2008.

[41] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP '91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 1991.

[42] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, 1996.

[43] Klaus Iglberger and Ulrich Rüde. C++ design patterns: The sandwich pattern. Technical Report 09-18, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2010.

[44] ISO/IEC. SQL/foundation. Standard 9075-2, 2003.

[45] Samuel N. Kamin. Research on domain-specific embedded languages and program generators. *Electronic Notes in Theoretical Computer Science*, 14:149–168, 1998.

[46] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for Lisp-like multi-staged languages. In *Conference Record of the 33rd Symposium on Principles of Programming Languages*, volume 41 of *ACM SIGPLAN Notices*, pages 257–268, January 2006.

[47] Gavin King et al. *Hibernate Reference Documentation*, 3.6.0 edition, 2010.

[48] Thomas Kistler and Michael Franz. A tree-based alternative to java byte-codes. *International Journal of Parallel Programming*, 27 (1):21–33, February 1999.

[49] Robert Klepper and Douglas Bock. Third and fourth generation language productivity differences. *Communications of the ACM*, 38(9):69–79, September 1995.

[50] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the conference on LISP and functional programming*, pages 151–161, 1986.

[51] Dinesh Kulkarni, Luca Bolognese, Matt Warren, Anders Hejlsberg, and Kit George. *LINQ to SQL: .NET Language-Integrated Query for Relational Data*. Microsoft, March 2007. URL `http://msdn.microsoft.com/en-us/library/bb425822.aspx`.

[52] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the international workshop on Types in languages design and implementation*, volume 38 of *ACM SIGPLAN Notices*, pages 26–37, 2003.

[53] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ninth international conference on Functional programming*, volume 39 of *ACM SIGPLAN Notices*, pages 244–255, 2004.

[54] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the tenth international conference on Functional programming*, volume 40 of *ACM SIGPLAN Notices*, pages 204–215, 2005.

[55] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[56] Benoît Langlois, Consuela-Elena Jitia, and Eric Jouenne. Dsl classification. In *Proceedings of the 7th Workshop on Domain-Specific Modeling*, 2007.

[57] Julia Laval and Laurent Réveillère, editors. *Proceedings of the 2nd Workshop on Domains-Specific Program Development*, 2008. URL `http://www.labri.fr/perso/reveille/DSPD/2008/`.

[58] Cédric Lavanchy. DBC 2. Source code, 2006. URL `http://lampsvn.epfl.ch/svn-repos/dubochet/dbc/trunk/src/scala/dbc2/`.

[59] Cédric Lavanchy. Static type safety guarantees for the operators of a relational database querying system. Semester project report, June 2008.

[60] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, volume 35 of *ACM SIGPLAN Notices*, pages 109–122, 1999.

[61] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3:431–463, November 1993.

[62] David H. Lorenz and John Vlissides. Pluggable reflection: decoupling meta-interface and implementation. In *Proceedings of the 25th International Conference on Software Engineering*, pages 3–13, 2003.

[63] Ole Lehrmann Madsen. Virtual classes: a powerful mechanism in object-oriented programming. In *Conference proceedings on Object-oriented programming systems, languages and applications*, October 1989.

[64] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *ECOOP 2008*, number 5142 in Lecture Notes in Computer Science, pages 260–284, July 2008.

[65] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[66] Martin Odersky. *The Scala Language Specification, Version 2.8*. EPFL, November 2010.

[67] Martin Odersky and Lex Spoon. The Scala 2.8 collections API. Online document, September 2010. URL `http://www.scala-lang.org/docu/files/collections-api/collections.html`.

[68] Martin Odersky, Vincent Cremet, Christing Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. Technical Report IC/2002/070, École polytechnique fédérale de Lausanne, 2002.

[69] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

[70] *Pro*C/C++ Programmer's Guide (10g Release 2)*. Oracle, June 2005.

[71] André Pang, Don Stewart, Sean Seefried, and Manuel M. T. Chakravarty. Plugging Haskell in. In *Proceedings of the workshop on Haskell*, pages 10–21, 2004.

[72] Emir Pašalić and Nathan Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*, pages 136–167, 2004.

[73] Tiark Rompf. Virtualised language constructs. In Scala 2.9 prerelease code, 2010.

[74] Tiark Rompf. Type safe, context free, embedded languages. Personal communication, September 2010.

[75] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering*, pages 127–136, 2010.

[76] Matti Rossi, Juha-Pekka Tolvanen, Jonathan Sprinkle, and Steven Kelly, editors. *Proceedings of the 9th Workshop on Domain-Specific Modeling*, 2009.

[77] Matti Rossi, Juha-Pekka Tolvanen, Jonathan Sprinkle, and Steven Kelly, editors. *Proceedings of the 10th Workshop on Domain-Specific Modeling*, 2010.

[78] Michel Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, EPFL, 2005.

[79] Ulrik P. Schultz, Julia L. Laval, and Charles Consel. Automatic program specialization for java. *Transactions on Programming Languages and Systems*, 25(4):453–499, 2003.

[80] Ulrik Pagh Schultz. Partial evaluation for class-based object-oriented languages. In *Proceedings of the Second Symposium on Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 173–197, 2001.

[81] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002.

[82] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing as staged type inference. In *Proceedings of the 25th symposium on Principles of programming languages*, pages 289–302, 1998.

[83] Olin Shivers. A universal scripting framework, or Lambda: the ultimate "little language". In *Concurrency and Parallelism, Programming, Networking, and Security: Proceedings of the Second Asian Computing Science Conference*, pages 254–265, 1996.

[84] Kamil Skalski, Michal Moskal, and Pawel Olszta. Meta-programming in Nemerle. electronic document, 2006.

[85] Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of the 11th symposium on Principles of programming languages*, pages 23–35, 1984.

[86] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[87] Walid Taha. Domain-specific languages. In *Internation Conference on Computer Engineering and Systems*, pages xxiii–xxviii, November 2008.

[88] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248: 211–242, October 2000.

[89] Bruce Trask and Angel Roman. Using domain specific modeling in developing software defined radio components and applications. In *Proceedings of the 1st Workshop on Domain-Specific Program Development*, pages 27–30, 2006.

[90] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *Transactions on Programming Languages and Systems*, 30(6):1–37, October 2008.

[91] David Ungar, Adam Spitz, and Alex Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion to the 20th conference on Object-oriented programming, systems, languages, and applications*, pages 11–20, 2005.

[92] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. Technical Report SEN-R0032, Centrum voor Wiskunde en Informatica, 2000.

[93] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Object oriented methods for interoperable scientific and engineering*, pages 286–295, 1999.

[94] Jos Warmer. A model driven software factory using domain specific languages. In 4530, editor, *Model Driven Architecture—Foundations and Applications*, Lecture Notes in Computer Science, pages 194–203, 2007.

[95] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of the conference on Programming language design and implementation*, volume 28 of *ACM SIGPLAN Notices*, pages 156–165, 1993.

[96] David S. Wile. Supporting the DSL spectrum. *Journal of Computing and Information Technology*, 9(4):263–287, 2001.

[97] Moshé M. Zloof. Query by example. In *National Computer Conference*, American Federation of Information Processing Societies Conference Proceedings, pages 431–438, May 1975.

Curriculum Vitæ

Mr Gilles Dubochet

Born on 2 September 1980 in Heidelberg, Germany
Swiss Citizen of Montreux

Education

École Polytechnique Fédérale de Lausanne:

2005–2011 Docteur ès sciences (PhD)
2003–2005 Master of Science, Computer Science
2000–2004 Diplôme d'ingénieur informaticien

École des Arches, Lausanne:

1999–2000 Maturité fédérale scientifique

École Rudolf Steiner, Lausanne:

1995–1999 Diplôme de culture générale

Notable Experiences

2005–2011 Member of the Scala development team
2005–2010 Teaching assistant
2005 Master project, University of Edinburgh
2001–2004 Elected town councillor, Morges
2001–2004 Co-director of VnV software