

Type-Safe Metaprogramming and Compilation Techniques For Designing Efficient Systems in High-Level Languages

Présentée le 25 novembre 2020

à la Faculté informatique et communications
Laboratoire de théorie et applications d'analyse de données
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Lionel Emile Vincent PARREAUX

Acceptée sur proposition du jury

Prof. V. Kuncak, président du jury
Prof. C. Koch, directeur de thèse
Dr F. Pottier, rapporteur
Prof. O. Kiselyov, rapporteur
Prof. M. Odersky, rapporteur

*The reasonable man adapts himself to the world;
the unreasonable one persists in trying to adapt the world to himself.
Therefore all progress depends on the unreasonable man.*
— George Bernard Shaw

The limits of my language mean the limits of my world.
— Ludwig Wittgenstein

Acknowledgements

I would first like to thank my thesis supervisor Christoph Koch for his patience and understanding, for believing in me from the start and letting me work on what impassioned me most, and for pushing me to continue my career in academia. Along with Val Tannen, Christoph helped me navigate the research job market; I would have never secured such opportunities without his dedicated advice and guidance. I am also grateful to the members of my thesis committee, and in particular to François Pottier for his incredibly detailed feedback on my thesis, and to Oleg Kiselyov for his humbling but also illuminating criticism.

I would like to thank the people with whom I had the chance of collaborating during my PhD: Amir Shaikhha, my office mate, with whom I had many captivating technical conversations, and who kick-started my research; Antoine Voizard, my long-time friend and long-distance collaborator; Simon Peyton Jones, who took interest in some of my outlandish ideas, which are still to come to fruition; Aleksander Boruch-Gruszecki, whose obsession with Common Lisp continues to defy my understanding; Paolo G. Giarrusso, who regularly enlightened me with his vast knowledge of programming language research; and Viktor Kunčak, who lured me into the PhD program in the first place. I would also like to thank the other people who provided helpful suggestions and feedback on my various paper ideas and drafts, and on the design of Squid: Samuel Grütter, Vlad Ureche, Georg S. Schmid, Milos Nikolic, Daniel Lupei, Dmitry Petrashko, Manohar Jonnalagedda, Sandro Stucki, Aggelos Biboudis, Vojin Jovanovic, Eugene Burmako, and Martin Odersky.

I would like to thank my lab mates for contributing to the great working environment at EPFL: Sachin, Milos, Aleksandar, Mohammad, Mohammed, Amir, Yannis, Daniel, and Immanuel, as well as Simone for taking care of the lab.

Last but not least, I would like to thank Ana for providing some stability in my life and for always pushing me to improve; my father, without whom I would probably not have discovered my vocation and started an academic career; and my mother for never failing to support me.

Lausanne, November 19, 2020

L. P.

Abstract

Software engineering practices have been steadily moving towards higher-level programming languages and away from lower-level ones. Higher-level languages tend to greatly improve the safety and maintainability of software systems, because they handle various implementation details automatically, allowing programmers to focus on their problem domains.

However, the gains offered by higher-level languages are often made at the cost of reduced performance — these languages usually consume more memory, run more slowly and require expensive garbage-collecting runtime systems, a trend which has been worsening with the increasing adoption of functional programming in the industry.

Modern programmers are thus faced with a dilemma: should they favor safety, productivity, and lower maintenance costs, or should they focus on performance instead?

The central idea behind the present thesis is to solve this dilemma by making simultaneous advances in type systems, metaprogramming, and compiler technology. In particular, we study various metaprogramming techniques based on flexible statically-typed quasiquotation, which enable domain experts to safely define their own domain-specific optimizers and compilers. This way, developers can focus on programming with high-level domain abstractions, while having these abstractions optimized and compiled away automatically.

We present the design and implementation of Squid, a metaprogramming framework which augments the Scala programming language with novel multi-staged programming capabilities. Along with Squid, we present several application examples, including a polymorphic yet efficient library for linear algebra, a stream fusion engine which improves on the state of the art, a demonstration of query compilation by rewriting, a multi-stage SQL database system prototype, and two embedded Scala domain-specific languages to express, optimize, and compile language-integrated queries.

Résumé

Les pratiques du génie logiciel s'orientent progressivement vers les langages de programmation de haut niveau, et s'éloignent des langages de plus bas niveau. Les langages de haut niveau ont tendance à améliorer considérablement la sécurité et la maintenabilité des systèmes logiciels, car ils traitent automatiquement les différents détails d'implémentation, ce qui permet aux programmeurs de se concentrer sur leurs domaines d'application.

Cependant, les gains offerts par les langages de haut niveau sont souvent réalisés au prix d'une réduction des performances — ces langages consomment généralement plus de mémoire, fonctionnent plus lentement et nécessitent des systèmes d'exécution coûteux nécessitant des ramasse-miettes, une tendance qui s'est aggravée avec l'adoption croissante de la programmation fonctionnelle en entreprise.

Les programmeurs modernes sont donc confrontés à un dilemme : doivent-ils privilégier la sécurité, la productivité et la réduction des coûts de maintenance, ou doivent-ils plutôt se concentrer sur les performances ?

L'idée centrale de cette thèse est de résoudre ce dilemme en faisant des progrès simultanés dans les systèmes de types, la métaprogrammation et la technologie des compilateurs. En particulier, nous étudions diverses techniques de métaprogrammation basées sur une *quasiquotation* flexible et statiquement typée, qui permettent aux experts du domaine de définir en toute sécurité leurs propres optimiseurs et compilateurs spécifiques au domaine. De cette façon, les développeurs peuvent se concentrer sur la programmation avec des abstractions de haut niveau, tout en étant assuré que ces abstractions sont optimisées et compilées automatiquement.

Nous présentons la conception et l'implémentation de Squid, un *framework* de métaprogrammation qui augmente le langage de programmation Scala avec de nouvelles capacités de programmation dite multi-étapes (*multi-stage programming*). En plus de Squid, nous présentons plusieurs exemples d'application, y compris une bibliothèque polymorphe mais efficace pour l'algèbre linéaire, un moteur de fusion de flux qui améliore l'état de l'art, une démonstration de compilation de requêtes par réécriture, un prototype de système de base de données SQL multi-étapes, et deux langages intégrés à Scala pour exprimer, optimiser et compiler des requêtes intégrées au langage.

Contents

Acknowledgements	i
Abstract (English/Français)	iii
Introduction	1
1 Statically-Typed Code Manipulation with Analytic Quasiquotes	13
1.1 Introduction	14
1.1.1 Basics of Quasiquotation	14
1.1.2 Analytic Quasiquotes	15
1.1.3 Statically-Typed Quasiquotes	15
1.1.4 Quasiquotes in Various Languages	16
1.1.5 Best of Both Worlds	16
1.2 Expressing Code Manipulation	17
1.2.1 Explicit Approaches	19
1.2.2 Existing Scala Quasiquotes	23
1.2.3 Limitations of Scala Reflection Quasiquotes	25
1.3 Code Manipulation with Squid Quasiquotes	28
1.3.1 Basics of Squid Quasiquotes	28
1.3.2 Pattern Matching and Rewriting	29
1.3.3 Type Representation Implicits	29
1.3.4 Matching and Extracting Unknown Types	30
1.3.5 Nonlinear Pattern Variables	33
1.3.6 Cross-Quotation References	33
1.3.7 Cross-Stage References and Cross-Stage Persistence	33
1.3.8 Runtime Compilation and Cross-Stage Persistence	34
1.3.9 Automatic Function Lifting and Unlifting	35
1.3.10 Higher-Order Pattern Variables	35
1.3.11 Code Combinators	36
1.3.12 Call-By-Name Reduction Example	36
1.4 Safety Properties of Squid Quasiquotes	38
1.4.1 Hygiene	38
1.4.2 Scope Safety	39
1.4.3 Type Safety	39

Contents

1.4.4	GADT Reasoning	39
1.4.5	Pattern Matching Exhaustiveness	40
1.4.6	Safety of Rewriting	40
1.5	Example: A Quoted ANF Conversion	41
1.6	Type-Safe & Hygienic Macros for Scala	44
1.7	Related Work	44
1.7.1	Existing Quasiquotation Systems	44
1.7.2	Unification of Runtime and Compile-time Metaprogramming	46
1.7.3	Type-Safe Code Manipulation	47
1.7.4	Program Transformation	47
2	Application: A Polymorphic Yet Efficient Linear Algebra Library	49
2.1	Introduction	49
2.2	Motivation	51
2.3	PILATUS Design	53
2.3.1	Tagless Final	53
2.3.2	Semi-Ring and Ring	55
2.3.3	Module	56
2.3.4	Linear Map	57
2.3.5	Pull Array and Control-Flow Constructs	58
2.4	Matrix Algebra	59
2.4.1	Vector: Module + Pull Array	59
2.4.2	Matrix: Linear Map + Vector	60
2.4.3	Putting It All Together	61
2.5	Interpreted Languages	62
2.5.1	Standard Matrix Algebra	62
2.5.2	Graph DSL for Reachability and Shortest Path	64
2.5.3	Probabilistic Linear Algebra Language	66
2.5.4	Differentiable Linear Algebra DSL	69
2.6	Staging and Optimisation	71
2.6.1	Augmented Multi-Stage Programming	71
2.6.2	Staging PILATUS	71
2.6.3	Staged Representation Optimisations	73
2.6.4	Algebraic Optimisations	73
2.6.5	Fixed-Size Matrix DSL	75
2.6.6	Fused DSL	76
2.7	Evaluation	77
2.8	Related Work	78
2.8.1	Linear Algebra Languages and Libraries	78
2.8.2	Deforestation and Array Fusion	78
2.8.3	Automatic Differentiation and Differentiable Programming	79
2.8.4	Probabilistic Programming	79

2.9	Conclusions	80
3	The Modular Implementation of Squid	81
3.1	Introduction	81
3.2	The Intermediate Representation Base	82
3.3	Closed Worlds	83
3.4	Language Virtualization	84
3.5	Open Worlds	85
3.6	Support for IR Manipulation	85
3.7	Intermediate Representation Reinterpretation	87
3.8	One Interface to Rule Them All	87
3.9	Implementation of Squid Quasiquotes in Scala	89
3.9.1	Compilation of Squid Quasiquotes	89
3.9.2	Cross-Quotation References	91
3.9.3	Required Properties of the Macro System	93
3.9.4	Use of Runtime Reflection and Metaprogramming	93
3.10	Related Work	94
3.10.1	Quasiquotes for Domain-Specific Languages	94
4	Optimizing High-Level Libraries with Quoted Staged Rewriting	97
4.1	Introduction	98
4.1.1	Staging and Extensible Compilers	98
4.1.2	User-Defined Rewriting	99
4.1.3	Quoted Staged Rewriting	100
4.2	Multi-Stage Programming Limitations Exemplified	101
4.2.1	Staging the Power Function	101
4.2.2	New Optimization Opportunity	101
4.2.3	Limitations of Staging	103
4.3	Quoted Staged Rewriting	105
4.3.1	Rewriting <code>Math.pow</code>	105
4.3.2	Extending the Rewriting	106
4.3.3	Hybrid Approaches and Online Rewriting	107
4.3.4	Guarantees and Control	107
4.3.5	Modularity of Rewritings	108
4.3.6	Composing Uses of QSR Libraries	109
4.3.7	Optimizing Existing Libraries	109
4.4	Enabling Quoted Staged Rewriting	110
4.4.1	Effect-Sensitive A-Normal Form (ANF)	110
4.4.2	Effect System	111
4.4.3	Scalability of Code Pattern Matching	111
5	Application: A New Approach to Stream Fusion	113
5.1	Previous Approaches	113

5.2	Stream Fusion by CPS and Inlining	114
5.3	The Problem with flatMap	117
5.4	Enabling More Fusion by QSR	119
5.5	Correctness of the Stream Fusion Scheme	122
5.6	Extensibility of Optimizations	123
5.7	When Everything Else Fails — Fusing flatMap the Hard Way	124
5.8	Evaluation	127
5.8.1	Performance	127
5.8.2	Productivity	128
5.9	Conclusion	129
6	Improved Safety and Expressivity for Analytic Metaprogramming	131
6.1	Introduction	132
6.1.1	Motivating Example	132
6.1.2	Limitations of Higher-Order Abstract Syntax	133
6.1.3	Non-lexically-scoped Open Code Manipulation	133
6.1.4	Early Example of Rewriting	134
6.2	Presentation of Contextual Squid	135
6.2.1	Handling of Open Code in Contextual Squid	135
6.2.2	Rewrite Rules and Polymorphism	137
6.2.3	Fixed Point Rewritings	138
6.2.4	Free Variables and Substitution	139
6.2.5	Speculative Rewrite Rules	139
6.2.6	Motivating Example: Array of Tuples Optimization	139
6.3	Formalization of the Core Language	142
6.3.1	Syntax	142
6.3.2	Type System	145
6.3.3	Operational Semantics	147
6.3.4	Soundness of λ^{f}	152
6.4	Implementation in Scala	159
6.5	Application: Query Compilation By Rewriting	161
6.5.1	Systems as multi-level DSLs	161
6.5.2	Schema Specialization	162
6.5.3	Row-to-Column Store Transformer	163
6.6	Related Work	164
6.7	Conclusion	166
7	Hygienic Scope Polymorphism	167
7.1	Introduction	168
7.2	Metaprogramming Hygiene Beyond Macros	172
7.2.1	Hygiene Via the Type System	173
7.2.2	Naive Interpretation of Context Polymorphism	174
7.3	A Negative Result: No Hygiene With Plain Names	176

7.3.1	Core Problem	176
7.3.2	Reified Context Parameters	177
7.3.3	Reified Weakening	178
7.3.4	Context Evidence Opacity and Transparency	180
7.3.5	A Problematic Program	181
7.4	Hygiene Via Affine First-Class Bindings in $\lambda^{[\alpha]}$	182
7.4.1	First-Class Bindings in Squid	182
7.4.2	Presentation of $\lambda^{[\alpha]}$	184
7.4.3	Soundness	190
7.4.4	Straightforward Extensions	190
7.5	Implementation in Squid	191
7.6	Example Applications	193
7.6.1	Bindings reversal	193
7.6.2	<i>Encoding</i> Cross-Stage Persistence for a Staged Database	194
7.6.3	A safer take on flatMap streamlining	195
7.7	Related Work	197
8	Multi-Stage Programming in the Large with Staged Classes	199
8.1	Introduction	199
8.2	Presentation of Staged Classes	202
8.2.1	Classes in Scala	202
8.2.2	The Vector Class, Staged	203
8.2.3	Staged Class Instantiation	204
8.2.4	Generative Programming to Avoid Repetition	206
8.2.5	Generalizing the Vector Arity	206
8.2.6	Generalizing the Element Type	207
8.2.7	Direct and Staged Inheritance	209
8.2.8	Staged Class Caching	210
8.2.9	Generic Methods	210
8.2.10	Putting It All Together	211
8.3	Use Case: Typed Type Providers	211
8.3.1	An Embedded DSL for Record Type Providers	213
8.3.2	Implementing the Type Provider DSL	214
8.3.3	Type Provision From Data Samples	216
8.3.4	Evaluation	217
8.4	Related Work	218
8.5	Conclusion	220
9	Application: A Staged Database Compiler	221
9.1	Motivation	221
9.2	Architecture of the Staged Database System	222
9.2.1	Specialized Container Classes	223
9.2.2	Column Store Meta-Container Class	223

9.2.3	Loading and Emitting Data Efficiently	225
9.3	Compiling Queries On The Fly	226
9.3.1	An Additional Stage for Compiling Queries	227
9.4	An Embedded DSL for Data Definitions and Queries	229
9.4.1	Shallow DSL	230
9.4.2	Internal Representation of the Database	233
9.4.3	Query Representation	234
9.4.4	Query Lifting	237
9.5	Basic Optimization and Planning for Queries	239
9.5.1	Query Rewriting	239
9.5.2	Query Plans	240
9.5.3	Query Planning	243
9.6	Evaluation	247
9.6.1	Related Work	248
9.7	Conclusion	249
10	Comprehending Monoids with Class	251
10.1	Background on Comprehension	251
10.1.1	Origins	251
10.2	Comprehension for Queries	252
10.3	Why Monoid Comprehension?	253
10.3.1	Semantics of List and Monad comprehension	253
10.3.2	Embedding Monoid comprehension in Haskell	254
10.3.3	Semantics of Monoid comprehension	254
10.3.4	Encoding	255
10.3.5	Space Efficiency	256
10.4	SQL-style Grouping and Ordering	256
10.4.1	Grouping in monad comprehension	257
10.4.2	Grouping in monoid comprehension	258
10.4.3	Performance of grouping	258
10.4.4	Generality of grouping	259
10.4.5	Ordering	260
10.5	Conclusions on Monoid Comprehension in Haskell	261
10.6	Generalized Monoid Comprehension in Scala	261
10.6.1	The Full Monoid Comprehension Calculus	262
10.6.2	Semigroups and Canonical Monoids	262
10.6.3	Heterogeneous Collection Types	263
10.7	Optimizing Monoid Comprehension Queries with Squid	264
10.7.1	Motivating Example	264
10.7.2	Optimization Approach	265
10.7.3	Deeply Embedding Monoid Comprehensions	266
10.7.4	Query Rewriting and Planning	267

Conclusions and Future Work	269
A Improved GADT Reasoning in Scala	271
A.1 Introduction	272
A.2 Closed GADTs in Core Scala and DOT	273
A.2.1 Encoding of ADTs and Pattern Matching	274
A.2.2 GADTs and Object-Oriented Languages	274
A.2.3 Existential types and Subtyping Proofs	275
A.2.4 Closed GADT Encoding in Scala	275
A.2.5 Core Scala and DOT	277
A.2.6 Closed GADT Encoding in Core Scala	277
A.2.7 Summary	279
A.3 Open GADTs	279
A.3.1 Class Instance Matching	279
A.3.2 Understanding an Old Paradox	280
A.3.3 Solution: Invariant Inheritance	281
A.3.4 Type Parameters as Members	281
A.4 Further Work on GADTs in Scala	283
A.5 Conclusion	283
B Complete Encoding of GADT in pDOT	285
C Organization of the Streams Optimizer	289
D Code of the Microbenchmarks	291
 Bibliography	 299
Curriculum Vitae	323

Introduction

Software engineering practices have been steadily moving towards higher-level programming languages and away from lower-level ones. Higher-level languages tend to greatly improve the safety, productivity, and maintainability of software systems, because they handle various implementation details automatically, allowing programmers to focus on their problem domains.

However, the gains offered by higher-level languages are often made at the cost of reduced performance — these languages usually consume more memory, run more slowly and require expensive garbage-collecting runtime systems. Today, we run JavaScript on smart watches, reducing their battery lives, and we have machines in data centers churn through gigabytes of heap-allocated objects, wasting energy. This trend has been worsening with the increasing adoption of functional programming in the industry, resulting in beautifully-maintainable codebases which run with suboptimal performance characteristics.

Modern programmers are thus faced with a dilemma: should they favor productivity and lower maintenance costs, or should they focus on performance instead?

The main idea behind my thesis is that we can help solve this dilemma by making simultaneous advances in type systems, metaprogramming, and compilers technology.

Research Problem

Programming abstractions are one of the main culprits for the inefficiency of high-level programming languages. Layers of abstractions help modularize and encapsulate program components, but they tend to introduce some costs at run time, and to make analyzing and optimizing programs much harder for compilers and runtime systems.

Removing abstractions automatically

On the other hand, high-level programs can often be made as efficient as lower-level ones if we have the tools to remove these high-level abstractions, lowering them into specialized constructs which contain fewer indirections, exhibit more controlled memory allocations, and do not get in the way of compiler optimizations.

Introduction

Moreover, by removing abstractions *automatically* from user programs, one can retain the safety and maintainability of writing high-level code, but without paying for it at runtime.

However, removing abstractions automatically is no easy feat; previous experience in the field has shown that fully-automatic and general solutions are intractable and do not scale. Best-effort approaches to general-purpose compiler optimization usually fall short — they are far from unlocking the level of performance that could be obtained from comparable low-level code written by expert programmers. Cohen et al. [2006] and many others made this observation more than a decade ago; yet, despite impressive progress in compilation techniques for various specific domain, the general situation has remained essentially unchanged to this day.

In other words, the proverbial “sufficiently-smart compiler,” which would be able to automatically remove all overhead introduced by code abstractions, does not exist.¹

Tradeoffs in abstraction removal

Therefore, if we want to write high-level programs with the performance characteristics of lower-level approaches, we need to compromise either on *full automation* or on *full generality*, if not both. For instance, we may require users to explicitly *guide the compiler* towards efficient implementation strategies, or we may have them use *restricted sublanguage* in which more assumptions can be made, to be leveraged by specialized optimizers.

Importantly, these compromises do not stand in the way of our original vision, as long as they do not jeopardize the safety and modularity of software artifacts, as well as the productivity of developers — the goal is to retain most of the advantages of high-level languages while favoring approaches that allow for more efficient compilation and execution. The modularity aspect is key here: it would be highly unsatisfactory if, for example, user-specified optimization strategies silently broke down when the code is refactored and modularized (as is true of many existing approaches based on optimization hints for the compiler).

Research Approach

In the present thesis, we focus on approaches which let programmers safely define their own specialized optimizers and compilation strategies, to apply on their high-level domain-specific languages (DSL) and libraries.

The typical intended workflow is that performance experts (called **DSL designers**) define domain-specific languages and libraries together with strategies for optimizing and compiling them efficiently, and **DSL users** (who need not have the same technical knowledge as DSL

¹The nonexistence of such a universal optimizer for a general-purpose programming language is in fact easily derived from Rice's theorem. One could argue that *most* of the desired optimization could still be achieved with the help of heuristics; in practice however, optimizers quickly run into limitations due to the algorithmic complexity of analyzing programs precisely, the loss of precision often resulting in big losses in optimization opportunities.

designers) can then write efficient programs based on these abstractions. We are in particular interested in approaches which retain the advantages of high-level languages, including safety and productivity, not only for DSL users but *also for DSL designers*.

The task of DSL designers requires writing programs which manipulate other programs, an activity known as **metaprogramming**. To permit expressive metaprogramming techniques without compromising the safety and modularity of software systems, we explore different **type systems** which ensure that metaprograms are well-behaved.

Moreover, non-trivial program optimizations are greatly facilitated and made practical by using advanced intermediate representations of programs; therefore, we also explore ways in which we can reconcile type-safe metaprogramming with the **compiler technology** to enable such representations.

The research literature on these different techniques has a long and rich history, of which we attempt to make a (necessarily incomplete) summary in the next section.

High-Level Background

Abstractions and Metaprogramming. The idea of removing the overhead of abstractions using metaprogramming techniques is an old one. It dates back at least to the work of Ken Kennedy, who coined the phrase “abstraction without guilt” – which later morphed into “abstraction without regret” [Rompf et al., 2014, Koch, 2014]. A novelty of the present thesis is to focus on enhancing at the same time the *safety* **and** *expressiveness* of such metaprogramming techniques, whereas previous approaches were often lacking in either category, as we shall discuss later.

Embedded domain-specific languages (EDSL). An EDSL is a domain-specific language which is defined using the constructs of an existing language — the so-called *host language* [Hudak, 1996]. This approach has seen some success in expressive languages with a flexible syntax such as Haskell [Axelsson et al., 2010, Najd et al., 2016, Hudak, 1996] and Scala [Rompf and Odersky, 2010, Lee et al., 2011, Ofenbeck et al., 2013]. The main advantage of EDSLs is that they can reuse the facilities of the host language [Oliveira et al., 2009], such as its parser, type system, integrated development environments, etc.

In the context of embedded domain-specific languages, program optimizations may be classified into two categories: *generic*, and *domain-specific*.

Generic optimizations. These optimizations operate on the basic constructs of the *host* language, and often work directly with specialized intermediate representation for that language. Examples of such optimizations are inlining [Chang and Hwu, 1989], common-subexpression elimination [Rosen et al., 1988], and dead-code elimination [Knoop et al., 1994].

Domain-specific optimizations. These optimizations operate on the level of specific *DSL constructs* defined within the host language (as opposed to language constructs themselves). They work by transforming certain code patterns into patterns expected to be more efficient, relying on domain-specific knowledge that generic optimizers do not necessarily have access to [Oliveira et al., 2009, Mernik, 2012]. Examples of such optimizations are data representation specializations [Ureche et al., 2015, Mernik, 2012] and list fusion [Gill et al., 1993].

Applying domain-specific optimizations is often crucial to the performance of DSLs. For example, DSLs for digital signal processing, image processing, and numerical computing like Feldspar [Axelsson et al., 2010], Spiral [Puschel et al., 2005], Halide [Ragan-Kelley et al., 2013], and Liszt [DeVito et al., 2011] rely on this type of very specific optimizations.

Compilers for general-purpose languages are usually limited to generic optimizations and do not provide facilities for programmers to add their own domain-specific ones, with a few exceptions (see *rewrite rules* below).

Deep EDSLs. Approaches to EDSL optimization have relied on *deeply* embedding EDSLs [Jovanovic et al., 2014]. This means that DSL constructs defined in the host language no longer evaluate their result directly, but instead create an intermediate representation of the DSL program, which can be optimized and partially evaluated, and in a later phase compiled and executed. Several approaches have been used to simplify the use of EDSLs. For example, type-based embedding [Rompf, 2016], as employed in Scala by Delite [Lee et al., 2011] and LMS [Rompf and Odersky, 2010], or in Haskell by Feldspar [Axelsson et al., 2010] and others.

Program generation. Program generation is almost as old as the discipline of programming itself, dating back to the days of Lisp and PL/I. It is ubiquitous in software engineering, and is often used for improving the performance of software systems. C++ templates [Vandevoorde and Josuttis, 2002], Lisp macros [Kohlbecker et al., 1986], and Template Haskell [Sheard and Jones, 2002] are example techniques for generating programs. The so-called *generative programming* approach has long been a staple of high-performance computing [Cohen et al., 2006], for instance with *active libraries* [Veldhuizen and Gannon, 1998] and *telescoping languages* [Chauhan and Kennedy, 2001].

Partial Evaluation. The goal of partial evaluation [Jones et al., 1993] is to evaluate in advance the parts of a program that are known statically, resulting in the generation of a “residual” program to compute the remaining dynamic parts. The residual program will usually execute faster than the original program, since it has less work to do. *Online* partial evaluation detects the static parts of programs on the fly, as they arise from previous transformations, while *offline* partial evaluation relies on a separate “binding-time analysis” pass which has to make conservative assumptions. Although the offline approach is less powerful, it is easier to implement and use in practice — this is especially true for more advanced applications, such as the so-called Futamura projections [Futamura, 1999]. There are three Futamura projections: the first projection consists in compiling a given program by specializing an interpreter for

that program; the second projection builds a compiler by specializing the specialized for the interpreter; finally, the third projection builds a compiler compiler by specializing the specialized... for itself. (Despite sounding crazy, this makes perfect sense.)

Multi-stage programming. Multi-stage programming (MSP) or just *staging* [Taha and Sheard, 1997] is a technique for specializing programs in a type-safe and modular way, using the full abstraction capabilities of a general-purpose host language. MSP lets programmers syntactically distinguish multiple stages of execution in their programs. At each intermediate stage, the program computes away what is known at this stage, and generates a new residual program meant to execute the next stage. The ultimate stage performs the task of the unstaged program, but in a more efficient way. MSP is a form of offline partial evaluation, with explicit annotations for binding-time analysis. It was introduced as a reaction to the unpredictability of original partial evaluation techniques (mentioned in the previous paragraph). In essence, MSP is used to define program generators that work by composing program fragments together in a type-safe way. MSP frameworks were developed for various programming languages, such as ML [Taha and Nielsen, 2003, Taha and Sheard, 1997, Taha, 1999], OCaml [Calcagno et al., 2003, Kiselyov, 2014], Scala [Rompf and Odersky, 2010, Rompf et al., 2013, Parreaux et al., 2017c,a], Haskell [Mainland, 2012], and even Java [Westbrook et al., 2010].

Extensible Compilers. Techniques inspired by MSP have been used to facilitate the definition of extensible compilers for performance-oriented DSLs and heterogeneous target platforms [Lee et al., 2011, DeVito et al., 2013, Puschel et al., 2005, Ofenbeck et al., 2013, Axelsson et al., 2010]. Generally speaking, these compilers reuse the frontend capabilities of their host (syntax and type system) but they convert programs into their own domain-specific intermediate representation (IR) which can be extended by DSL designers for custom optimization and compilation capabilities.

Quasiquotes. Pioneered in Lisp [Bawden, 1999] and later picked up for supporting MSP, *quasiquotes* are a convenient way of manipulating program fragments using the concrete syntax of the manipulated language. Quasiquotes act like quoted code templates that offer a way for program fragments to be composed (put together to form bigger programs) and decomposed (inspected and broken down into smaller parts).

Macros. Languages like Lisp, Scala [Shabalin et al., 2013], and Haskell [Sheard and Jones, 2002] provide *macros*, a way to have fragments of code execute at compilation time, manipulating the AST representations of user programs. Macro systems often rely on quasiquotes to make this process easier [Bawden, 1999]. More primitive macro systems, based on textual manipulation of the program's source code (so-called *macro preprocessors*), date back to the 1960s, in the context of languages like PL/I and C.

Inlining. Inlining [Chang and Hwu, 1989], copies the bodies of functions into their call sites, in order to optimize them with their surrounding environments. While instrumental in exposing

optimization opportunities [Leißa et al., 2015], inlining has to be performed carefully to avoid making it counter-productive. Indeed, not only can excessive inlining lead to code size explosion, but it can also lead to high-level optimization opportunities being missed, when these optimizations rely on detecting library usage patterns which could not be recovered easily from the inlining of their internal implementations. This is especially important for domain-specific optimizations expressed using *rewrite rules*.

Rewrite Rules. Rewrite rules like these of the Glasgow Haskell Compiler (GHC) [Peyton Jones et al., 2001] allow library authors to specify domain-specific optimizations like stream fusion [Coutts et al., 2007]. They have proven powerful but also brittle, as they are completely at the mercy of the compiler’s inlining heuristics, which do not always expose enough rewriting opportunities.

Supercompilation and distillation. The goal of these techniques [Turchin, 1996, Bolingbroke and Peyton Jones, 2010, Sørensen and Glück, 1995, Hamilton, 2007] is to optimize recursive programs by aggressive inlining steps (called *driving*) and generalization/folding steps. These techniques often generate lots of code duplication (sometimes prohibitively so), and they have not yet found their way into mainstream applications.

Normalizing intermediate representations (IR). Many IRs have been proposed for simplifying data-flow analysis and program optimization, such as SSA [Rosen et al., 1988], CPS [Appel, 1992, Kennedy, 2007], ANF [Flanagan et al., 1993], VSGD [Stanier, 2012, Reißmann, 2012], and the sea-of-IR-nodes [Click and Paleczny, 1995]. These IRs are crucial for detecting optimization opportunities and applying them effectively and efficiently in user programs, as working with plain abstract-syntax trees quickly becomes too limiting. In the context of functional programming language compilers, ANF-based and CPS-based IRs have traditionally been favored. There has been some contention in the community [Appel, 1992, Kennedy, 2007, Flanagan et al., 1993, Maurer et al., 2017] on whether ANF is sufficient to reap the advantages of the more complex CPS representation, and it seems like the debate is still ongoing.

We will review more specific background and related work within each chapter of the thesis, as the need arises.

Research Questions

The principal research question we aim to answer are the following:

- How to design expressive metaprogramming constructs to describe domain-specific optimizations of high-level programs in a safe way?
- How to make these metaprogramming constructs extensible and able to leverage existing compiler technology, like advanced intermediate representations?

ADT	algebraic data type
ANF	A-normal form
AST	abstract syntax tree
CPS	continuation-passing style
CSE	common subexpression elimination
DCE	dead code elimination
DSL	domain-specific languages
EDSL	embedded domain-specific languages
GADT	generalized algebraic data type
HOAS	higher-order abstract syntax
IGR	incremental graph reduction
IR	intermediate representation
LHS	left-hand side
MSP	multi-stage programming
PHOAS	parametric HOAS
QQ	quasiquote
QSR	quoted staged rewriting
RHS	right-hand side
redex	reducible expression

Table 1 – common abbreviations and shorthands used throughout the thesis.

- How to integrate such design into an existing programming language like Scala, leveraging its advanced type and macro systems, and avoiding extensive changes to its compilation process?
- How to extend the multi-stage programming (MSP) paradigm to make it applicable to more real-world use cases, taking it beyond its current niches?
- What does the design of efficient systems based on these approaches look like, and how to solve the challenges which arise from it in practice?

Preliminaries

Before explaining the contributions of this thesis, let us first go through a few preliminaries.

Terminology

Table 1 lists some common abbreviations and shorthands used throughout the thesis.

Static Types for Metaprogramming

In this thesis, we tackle the problem of statically typing *metaprograms*: programs that construct, deconstruct, rewrite and evaluate other programs. The famous motto of static typing coined by Milner [1978], *Well-typed programs cannot “go wrong”*, which means that types prevent the occurrence of runtime errors, can be adapted to our specific setting:

Well-typed metaprograms cannot “go wrong!”

In particular, well-typed metaprograms should not run into type mismatches and unbound variable errors at runtime, which could arise from erroneous uses of code manipulation constructs. This is what we mean when we refer to the *safety* of our metaprogramming primitives.

The Scala Language

Scala was originally a research language developed by Martin Odersky and his team at EPFL, but it has since become an important language in both research and industry. Scala has gained a significant and diverse user base, making it one of the most successful functional programming languages to date.

Most of the ideas and designs presented in this thesis are general and broadly applicable. However, we focus on examples using the Scala programming language (we also use some Haskell for illustration purposes in Chapter 10). Although we make a point of explaining every nonstandard feature of the language, some familiarity with Scala will likely help the reader understand our examples in their finest detail.

Scala Reflection Macros

Since its version 2.10, Scala has offered experimental support for an advanced macro system [Burmako, 2013] based on the so-called *Scala Reflection API* [Burmako, 2017a]. Scala’s macro system is atypical in that it expands macro invocations during the elaboration phase of the type checker, so that Scala macros can influence the processes of type inference and type checking. This can be used to extend Scala’s type system, providing much more power than the more common syntactic approaches to macro expansion.

Despite their experimental nature and their lack of stability, Scala macros have been adopted by large swathes of the ecosystem, and libraries built using Scala macros have become essential parts of the Scala landscape.

The Squid Framework

The main software artifact developed during my PhD is the **Squid** type-safe metaprogramming and extensible compilation framework, implemented as a macro-based extension to the Scala programming language. “Squid” stands for the approximate contraction of *Scala **quoted** DSLs*. The framework is open source and available online.²

Squid has already seen a number of users, including students from several universities. It was also used by my former colleague Amir Shaikhha, currently a lecturer at the University of Oxford, which resulted in the collaborative work presented in Chapter 2.

Dotty and Scala 3

Dotty is the name of the next-generation Scala compiler, which is being developed by Martin Ordersky and his team at EPFL. This compiler embodies the upcoming version 3 of the language, and is slated to replace the old Scala 2 compiler in the near future.

One of the most difficult aspects of porting existing Scala code to Dotty is that the old Scala macro facilities, which had always been experimental, are no longer available in Dotty. This decision was made because the old macro facilities were too unprincipled and were tied to the internals of the old Scala compiler. Consequently, considerable efforts have gone into designing a new macro system which can support the most important usages of existing Scala macros in the wild, while fixing their limitations.

After the publication of Squid, which featured the first quotation-based MSP system for Scala, similar MSP capabilities were added to Dotty to support the macro redesign [Stucki et al., 2018]. Moreover, several unique innovations of Squid, such as statically-typed analytic quasiquotes, have already made their way into Dotty’s new metaprogramming system.

Main Contributions

The main contributions of this thesis are the following:

- I present the design and implementation of Squid, a type-safe metaprogramming and extensible compilation framework, as a Scala macro-based library (Chapters 1 and 3).
- I explain some of the limitations of existing metaprogramming approaches; in particular, I discuss the limitations of multi-stage programming (MSP), and through Squid I show how to extend MSP in several directions: 1. by supporting the decomposition and analysis of program fragments (Chapter 1); 2. by using normalizing underlying intermediate representations to facilitate optimizations (Chapters 3 and 4); and 3. by adding, via

²Squid is available at <https://github.com/epfldata/squid>.

staged classes, the capability to manipulate and generate classes and modules which can be shared across a staged application (Chapter 8).

- I introduce an increasingly expressive type system to support the metaprogramming constructs of Squid corresponding to the aforementioned features (Chapter 6), culminating in the presentation of a system for type-safe and scope-safe analytic metaprogramming with hygienic manipulation of open program fragments and context polymorphism (Chapter 7). I provide a formalization of this type system and prove its soundness. This type system is reused to support the safety of staged classes (Chapter 8).
- The development of Squid and its type system have been motivated from the start by very practical metaprogramming use cases in the development of efficient systems. Throughout the thesis, I present various application examples of the demonstrated Squid capabilities, including a polymorphic yet efficient library for linear algebra (Chapter 2), a stream fusion engine improving on the state of the art (Chapter 5), a demonstration of query compilation by rewriting (Section 6.5), the design of a multi-stage SQL database system prototype along with its embedded Scala DSL frontend (Chapter 9), and a new embedded domain-specific language for better expressing and optimizing queries over collections of data (Chapter 10).

The contents of this thesis are in large parts derived from published and in-progress work in collaboration with Amir Shaikhha, Antoine Voizard, Aleksander Boruch-Gruszecki, Paolo G. Giarrusso, and Christoph Koch, my thesis supervisor. The corresponding published papers are listed below:

Squid: Type-safe, hygienic, and reusable quasiquotes (SCALA 2017) [Parreaux et al., 2017b]

Quoted Staged Rewriting: A practical approach to library-defined optimizations (GPCE 2017, Best Paper Award) [Parreaux et al., 2017a]

Unifying analytic and statically-typed quasiquotes (POPL 2018) [Parreaux et al., 2017c]

Comprehending monoids with class (TyDe 2018, Extended Abstract) [Parreaux and Koch, 2018]

Finally, a Polymorphic Linear Algebra Language (ECOOP 2019) [Shaikhha and Parreaux, 2019]

Towards improved GADT reasoning in Scala (SCALA 2019) [Parreaux et al., 2019]

Multi-stage Programming in the Large with Staged Classes (GPCE 2020) [Parreaux and Shaikhha, 2020]

Outline of the Thesis

We start by motivating and exploring Squid’s basic quasiquotation capabilities in **Chapter 1**, demonstrating the design of the first practical statically-typed quasiquotation system to allow code inspection while preserving type safety, scope safety, and hygiene.

I present an early application example in **Chapter 2**: the design of a high-level polymorphic library for linear algebra, which uses Squid to achieve the performance of specialized low-level implementations.

In **Chapter 3**, we discuss the modular implementation of Squid in *tagless-final* style (i.e., using *object algebras*), which enables the use of quasiquotation with different possible underlying intermediate representations.

In **Chapter 4**, we identify some limitations of multi-stage programming and propose a new approach called *quoted staged rewriting* (QSR). This approach can be used to define optimizations as rewrite rules using statically-typed analytic quasiquotes backed by a normalizing intermediate representation

As an application example, **Chapter 5** shows the design of a stream fusion approach using QSR which improves on the state of the art by being both simpler and more powerful.

In **Chapter 6**, we revisit the handling of bindings as shown in Chapter 1, finding it to be too restrictive; we present and formalize a new approach which allows users to more freely decompose and recompose bindings while still preserving scope safety and hygiene. Based on the new-found expressive power of these binding manipulation capabilities, I describe a new optimization technique called *speculative rewrite rules*. Speculative rewrite rules are used to design a query compiler implemented by rewriting, as an application example. However, the approach shown in this chapter is not final, and is only a stepping stone to the next chapter.

The system presented so far is *still* too restrictive as it has limited support for scope polymorphism. In **Chapter 7**, we tackle the question of enabling hygienic scope polymorphism in Squid. We devise a new formal calculus to solve this problem via a (lightweight) dependent affine type system. This new system is expressive enough to describe the stream fusion approach of Chapter 5 with minimal changes, removing its usage of unsafe escape hatches.

In **Chapter 8**, we identify further limitations of multi-stage programming which prevent it from being used “in the large,” on the level of modules and data structures, as opposed to single function bodies. We introduce *staged classes* as a way to manipulate class definitions as first-class constructs in a type-safe way. This allows generating efficient specialized modules and data structures which can be shared and reused across each staged application.

As an application example for staged classes, we show in **Chapter 9** the design of a staged SQL *database compilation* prototype which goes beyond the simple *query compilation* techniques demonstrated in previous work.

Introduction

As can be seen above, several application examples on which I have focused are related to database use cases. During my PhD, I often felt the need to embed query languages inside Scala or Haskell, and I have been looking for a query language most appropriate for such embedding (and corresponding optimization). This led me to revisit an old system, the *monoid comprehension calculus*, in the context of a functional language with type classes, which turned out to unleash the expressive power of monoid comprehension, making it a good frontend language for my database prototypes. The embedding of this query language in both Haskell and Scala is explained in **Chapter 10**, where we also briefly outline how to optimize it using Squid.

We then conclude and present future work.

Finally, in appendix Appendix A, we look at two particular typing problems which arose from quasiquotation in Squid, but which are more general and can be studied independently; they are the problems of reasoning about generalized algebraic data types (GADTs). We explore foundations for GADTs within Scala’s core type system (which is unique in that it supports both GADTs and subtyping), in order to guide a principled understanding of the required reasoning capabilities in Scala and Squid.

1 Statically-Typed Code Manipulation with Analytic Quasiquotes

Quasiquotation is a technique to construct and deconstruct program fragments using quoted code templates in which “holes” are left to be filled in later. This can greatly simplify the task of metaprogramming, because such *quasiquotes* hide the details of the representation in which programs are manipulated — quasiquotes present metaprogrammers with the concrete syntax of the language they seek to manipulate.

I argue that two main flavors of quasiquotes have existed so far: Lisp-style quasiquotes, which can both *construct* and *deconstruct* programs but may produce code that contains type mismatches and unbound variables; and MetaML-style quasiquotes, which rely on static typing to prevent these errors, but can only *construct* programs. The former has been used for designing expressive macros as well as program analysis and transformation systems, and the latter has been particularly useful for multi-stage programming and embedded domain-specific languages.

In this chapter, I describe a quasiquotation system which unifies Lisp-style and MetaML-style quasiquotes into a single framework called **Squid**, a library which extends Scala’s type system with statically-typed code manipulation capabilities. We show how to support the type-safe construction *and* deconstruction of program fragments. To the best of my knowledge, this is the first practical statically-typed quasiquotation system to allow code inspection.

Combining Lisp-style quasiquotes with static typing is challenging for multiple reasons. We should statically ensure *type safety* (no type mismatches), *scope safety* (no unbound variable), and *hygiene* (no name clashes) throughout our program manipulations. In particular, when pattern-matching on program fragments, we need the compiler to soundly reason about code patterns which can both uncover unknown types and refine known ones (GADT reasoning, which we explore in its more general form in Appendix A).

I demonstrate the use of Squid by implementing a statically-typed quoted β reduction and ANF conversion, and explain how Squid can be used as a type-safe and hygienic alternative to the current Scala macros.

1.1 Introduction

We start by recalling some background on quasiquotation, and classify the existing systems between two main paradigms: Lisp-style analytic quasiquotation, and MetaML-style statically-typed quasiquotation.

1.1.1 Basics of Quasiquotation

Our solution to implementing safe metaprogramming constructs is crucially based around the concept of code quasiquotes — or just *quasiquotes*: quoted code templates that offer a way for program fragments to be composed (put together to form bigger programs) and decomposed (inspected and broken down into smaller parts).

As a basic example, in Squid the value denoted by `code"2 + 2"` is *not* a string of characters, but an abstract syntax tree (AST) representing the expression $2 + 2$. These quasiquotes can be viewed as syntactic sugar for manipulating AST node constructors — indeed, one could express `code"2 + 2"` more explicitly as `IntAdd(Const(2), Const(2))`.

Within quasiquotes, it is possible to leave holes (also called *unquotes*, or *antiquotes*) to be filled in later. Holes are written `${...}` or just `$id` when `...` is a simple identifier `id`.

In expressions, holes enable code *insertion*: they are substituted with the provided code values. For example, in a context where `x = code"2"`, the expression `code"2 + $x"` is equivalent to `code"2 + 2"`. In patterns, holes enable code *extraction*: they pull code values out of the matched programs, making the result available to the right-hand side of the corresponding pattern matching branch.

As an example of code pattern-matching,¹ the following code:

```
code"print(27 + 1)" match {  
  case code"print($x)" => x  
}
```

extracts the code fragment passed to `print` in the original program `code"print(27 + 1)"`, and thus evaluates to `x = code"27 + 1"`.

In this work, we focus on the quasiquotation of code in the same language as the host language (the language in which code manipulation is done — here Scala). This is not a significant restriction in practice, as long as the host language is powerful enough to express the programs we want to manipulate. For example, many domain-specific languages (DSL) have been successfully embedded in expressive languages with a flexible syntax such as Haskell [Axelsson et al., 2010, Najd et al., 2016, Hudak, 1996] and Scala [Rompf and Odersky, 2010, Lee et al., 2011, Ofenbeck et al., 2013].

¹ Scala expression `s match {case p => e}` corresponds to SML's `case s of p => e` or Caml's `match s with p -> e`.

Code quasiquotation has been present in research and industry under two main flavors, which we will refer to as the *Analytic* and *Statically-Typed* flavors.

1.1.2 Analytic Quasiquotes

Analytic quasiquotes were pioneered in the context of Lisp, where source code is essentially made of arbitrarily-nested lists and symbols (S-expressions), which is also the native data structure format which programs manipulate.

For example, the Lisp function `(lambda (x) (print x))` which takes a parameter `x` and prints its value, can be represented with datum ``(lambda (x) (print x))` — the “back-tick” at the beginning indicates the start of a quasiquotation, to distinguish it from plain Lisp source code. Therefore, programs can naturally manipulate source code like any other data structure (code as data). In addition, the built-in `eval` function is used to interpret any datum as source code by executing it (data as code).

Antiquotation in Lisp is written with a comma, so expression ``(lambda (x) ,(id `x))` seen in Table 1.1 first executes the identity function `id` on the symbol ``x` returning ``x`, then places that code fragment into the bigger program fragment, constructing the code for another implementation of the identity function ``(lambda (x) x)`.

The fact that Lisp programs can naturally analyze (inspect) source code derives directly from the idea of code as data. Allowing quasiquotes in *pattern matching* is one convenient way to do such analysis, and is in fact standard in several dialects of Lisp, including Scheme [Bawden, 1999]. This is what motivates our terminology: these quasiquotes have *analytic* capabilities.

1.1.3 Statically-Typed Quasiquotes

Lisp is *dynamically typed*, which means that it does not statically prevent the occurrence of type mismatches and unbound variable references at runtime. For example, since the identifier `x` is a valid bit of Lisp program, `(eval `x)` is also valid but raises a runtime error in the style of “`x` is undefined and cannot be evaluated” unless it is executed in a context where some `x` is defined.

In contrast, in a *statically typed* programming language, type mismatches and undefined variable errors are never supposed to happen at runtime. The presence of an `eval` function in this setting makes the notions of code as data and data as code significantly harder to satisfy. Indeed, we must make sure that program fragments containing unbound references are never evaluated, and that all constructed programs are well-typed. We must reject programs such as `code" x ".run` (using Squid syntax, where method `run` evaluates the program fragments, just like `eval` in Lisp).

With MetaML, Taha and Sheard [2000] introduced statically-typed quasiquotes, allowing

the expression of type-safe program generators that cannot generate ill-typed or ill-scoped programs. In MetaML, the quotation `< x + 1 >` — similar to Lisp’s ``(+ x 1)` — is only valid if it is surrounded by a code fragment *at the same quotation level*² containing a binder for `x` with type `int`, so that `x + 1` will end up in a place where `x` is bound when the metaprogram executes. Together with antiquotes, written `~(. . .)`, we can rewrite in MetaML the Lisp example we saw in the previous section as `< fun x → ~(id < x >) >`, which evaluates to `< fun x → x >`.

MetaML historically faced some safety challenges. The first was to statically prevent the evaluation of open code (code that contains variables which have not yet been bound). For example, `< fun x → ~(run < x > ; < x >) >` should be rejected: `< x >` cannot be run as it has not yet been inserted into a context where `x` is bound. Another challenge was that of *scope extrusion*, where a piece of code *escapes* its enclosing scope by ways of imperative features such as mutable references, as in `< fun x → ~(some_ref := < x > ; < x >) >`. Two general approaches have been proposed to solve these problems; the first makes use of contextual types [Nanevski, 2002, Rhiger, 2005, Kim et al., 2006], where the environments that terms depend on are reflected in their types; the second uses *environment classifiers*, which abstract over these contexts [Taha and Nielsen, 2003] using type variables, which can be partially-ordered to reflect the structure of scopes [Kiselyov et al., 2016].

MetaML quasiquotes are not as expressive as analytic quasiquotes like those of Lisp, because all constructed code is viewed as a black box that cannot be inspected; in other words, these quasiquotes cannot be used in patterns,³ and it is not possible to use them to express program analysis or transformation algorithms.

1.1.4 Quasiquotes in Various Languages

Quasiquotes that belong to the analytic category include those of the Lisp family [Bawden, 1999], Stratego [Visser, 2002] and Scala reflection [Shabalin et al., 2013] (see Section 1.2.2). The safer, statically-typed category includes the quasiquotes of MetaML [Taha and Sheard, 2000], MacroML [Ganz et al., 2001], MetaOCaml [Taha, 2004] and Typed Template Haskell, a variant of Template Haskell [Sheard and Jones, 2002].

Table 1.1 summarizes the properties of some of these various systems, and is discussed further in the related work (Section 1.7).

1.1.5 Best of Both Worlds

In this chapter, I show how to combine the advantages of both flavors of quasiquotes into a unified framework, realized as Squid. Squid allows the construction and inspection of code fragments while ensuring that generated code is always well-typed and *usually* well-scoped.

²Notwithstanding the cross-stage persistence capability, which we will discuss later.

³Quasiquote patterns for MetaML were suggested by Sheard et al. [1999], but were neither implemented nor formalized.

1.2. Expressing Code Manipulation

	T	S	A	H	Syntax Example
Squid (This Chapter)	●	◐	◐	●	<code>code"(x: T) => \${ id(code"x") }"</code>
Scala-reflection QQ (1)	○	○	●	○	<code>q"(x: T) => \${ id(q"x") }"</code>
Scala-refl. reify/splice	◐	○	○	●	— cannot express open terms —
MetaOCaml (2)	●	◐	○	●	<code>.< fun x → .~(id .< x > .) >.</code>
Template Haskell (3)	○	◐	○	◐	<code>[\x -> \$(id [x])]</code>
Typed Template Haskell	●	●*	○	●*	<code>[\x -> \$\$ (id [x])]</code>
Stratego (4)	○	○	●	○	<code>[(x: int)=> ~(id ([x]))]</code>
Lisp/Scheme QQ (5)	○	○	●	○	<code>`(lambda (x) ,(id `x))</code>
λ^{\dagger} (Chapter 6)	●	●	●	●	<code>[$\lambda x:T.$ [id [$x:T$]]]</code>
MetaML Calculus (7)	●	◐	○	●	<code>< $\lambda x.$ ~(id < x >) ></code>
Rhiger's λ^{\dagger} (6)	●	●	○	●	<code>\uparrow ($\lambda x:T.$ \downarrow (id $\uparrow x$))</code>
Nanevski's v^{\square} (8)	●	●	◐	●	<code>let box u = id X in box $\lambda x.$ { $X \doteq x$ } u</code>

Table 1.1 – Comparison of quasiquotes in several systems. The criteria are whether they: statically ensure program fragments are *well-typed* and *well-scoped* (columns *T* and *S*, respectively); support *analysis* via pattern-matching (*A*); support *hygiene* (*H*); the asterisks indicate that although fully supported, the feature is not as flexible as in other approaches; see Sections 1.7.1 and 6.6 for a full discussion. References: (1) Shabalin et al. 2013; (2) Taha 2004; (3) Sheard and Jones 2002; (4) Visser 2002; (5) Bawden 1999; (6) Rhiger 2012b; (7) Taha and Nielsen 2003; (8) Nanevski and Pfenning 2005.

The scope safety guarantee mentioned in this chapter has the same limitations as in the early MetaML work, in that it only holds as long as metaprograms are pure and do not use `eval`. We will explore a more advanced contextual type system to address these limitations in Chapter 6.

Note that Squid quasiquotes focus on the *expression* side of Scala; they cannot manipulate class, method, object, or type definitions. This restriction is similar to other staging frameworks, such as MetaML [Taha and Sheard, 2000], MetaOCaml [Kiselyov, 2014], and LMS [Rompf and Odersky, 2010]. In contrast, the existing Scala 2 quasiquotes allow manipulating all Scala constructs, but with much weaker guarantees. In Chapter 8, we explore how to support the first-class representations of classes in a safe way, in order to support more powerful program analysis and generation strategies.

1.2 Expressing Code Manipulation

In this section, I describe some of the problems which arise in the context of analytic metaprogramming, reviewing the limitations of existing approaches in the context of Scala.

```
sealed abstract class Exp
case class Lit (value: Int)           extends Exp
case class Var (name: String)         extends Exp
case class Add (lhs: Exp, rhs: Exp)   extends Exp
case class App (fun: Exp, arg: Exp)   extends Exp
case class Fun (name: String, body: Exp) extends Exp
```

(a) Simple uni-typed abstract syntax tree (AST).

```
sealed abstract class Exp[T]
case class Lit      (value: Int)           extends Exp[Int]
case class Add      (lhs: Exp[Int], rhs: Exp[Int]) extends Exp[Int]
case class App[A, R] (fun: Exp[A => R], arg: Exp[A]) extends Exp[R]
case class Fun[A, R] (lam: Exp[A] => Exp[R]) extends Exp[A => R]
```

(b) AST based on a generalized algebraic data type (GADT) and using higher-order abstract syntax (HOAS) to represent bindings.

```
type Exp[T] = [V] => Term[T, V]
sealed abstract class Term[T, V](ty: Type[T])
case class Lit[V]      (value: Int)   extends Term[Int, V](IntType)
case class Var[A, V]   (value: V, tyA: Type[A])
                                extends Term[A, V](tyA)
case class Add[V]      (lhs: Term[Int, V], rhs: Term[Int, V])
                                extends Term[Int, V](IntType)
case class App[A, B, V] (fun: Term[A => B], arg: Term[A, V], tyB: Type[B])
                                extends Term[B, V](tyB)
case class Fun[A, B, V] (lam: V => Term[B, V], tyA: Type[A], tyB: Type[B])
                                extends Term[A => B, V](FunType(tyA, tyB))

sealed abstract class Type[T]
case object IntType           extends Type[Int]
case class FunType[A, B](tyA: Type[A], tyB: Type[B]) extends Type[A => B]
```

(c) GADT AST with *parametric* HOAS bindings (PHOAS) and internal typing.

Figure 1.1 – Outline of different possible AST implementations.

1.2.1 Explicit Approaches

Simple abstract syntax trees

The simplest possible way to represent and manipulate programs, in a functional language, is through an abstract syntax tree representation implemented with an algebraic data type.

Figure 1.1a shows the definition of an AST for lambda calculus augmented with integer literals and addition. Notice that all expression nodes have the same base type `Exp`, and that names are bound in `Fun` and referred to in `Var` as simple character strings.

In this representation, one step of call-by-name reduction (for example) may be written as follows:⁴

```
def reduce: Exp => Exp = {
  case Lit(v)           => Lit(v)
  case Var(n)           => Var(n)
  case Add(Lit(l), Lit(r)) => Lit(l + r)
  case Add(Lit(v), r)    => Add(Lit(v), reduce(r))
  case Add(l, r)         => Add(reduce(l), r)
  case App(Fun(n, b), a) => subst(n, a)(b)
  case App(f, a)         => App(reduce(f), a)
  case Fun(n, b)         => Fun(n, b)
}

def subst(name: String, arg: Exp): Exp => Exp = {
  case Lit(v)           => Lit(v)
  case Var(`name`)      => arg
  case Var(n)           => Var(n)
  case Add(l, r)         => Add(subst(name, arg)(l), subst(name, arg)(r))
  case App(f, a)         => App(subst(name, arg)(f), subst(name, arg)(a))
  case Fun(`name`, b)    => Fun(name, b) // `b` cannot refer to the variable
  case Fun(n, b)         => Fun(n, subst(name, arg)(b)) // unhygienic
}
```

Manipulating such an AST representation is error-prone mainly for three reasons:

- it is all too easy to construct *ill-typed* terms such as `App(Lit(1), Lit(2))`, as it is to make mistakes which can lead to such terms, for example by writing `App(reduce(a), a)` instead of `App(reduce(f), a)` in the `App(f, a)` case of `reduce`;
- it is all too easy to construct *ill-scoped* terms such as `App(Var("oops"), Lit(3))`, which

⁴In Scala, `{ case ... }` is the syntax of `PartialFunction` literals, but it can also be used as a shorthand syntax for `x => x match { case ... }` of type `A => B`. Also in Scala, patterns of the form ``v`` match any value equal to an existing `v` value (the ticks are used to distinguish these “equality patterns” from pattern variables).

could arise for instance if we had written `subst(name, arg)(b)` instead of `Fun(n, subst(name, arg)(b))` in the corresponding `subst` case;

- it is all too easy to manipulate name bindings *unhygienically*, i.e., to write program transformations which result in name clashes (unintended variable capture/shadowing); in fact, the `subst` implementation we’ve seen is unhygienic because it does not perform any capture avoidance: it may substitute a term `arg` within a context where its free variable occurrences get captured (which is not the intended semantics of substitution).

None of these problems are detected at the time the metaprogrammer defines their metaprogram; instead, they manifest themselves when the metaprogram is run on actual programs, resulting in the generation of nonsensical programs which do not compile, or — worse — the generation of programs which compile, but with the wrong runtime semantics.

Debugging this sort of problems by tracing back their root causes is hard and time-consuming, notably because they often occur far from these root causes, and because the code that is generated in many metaprogramming applications is inscrutable and not designed for human understanding.

Generalized algebraic data types (GADT)

To avoid some of these problems, the practice has been to *reflect* the type of each object-language term (a term in the language being manipulated) in the type of its corresponding AST node.

This can be done by using generalized algebraic data types (GADTs) [Xi et al., 2003, Cheney and Hinze, 2003, Kennedy and Russo, 2005], as shown in Figure 1.1b. Notice that `Exp` is now equipped with a type parameter that documents the type of the term it represents. While this generally improves the safety of AST manipulations, it also makes them slightly more cumbersome to write.

As we will see in Section 1.4 and in more depth in Appendix A, GADTs require some special reasoning capabilities from compilers to handle pattern matching adequately.

Higher-order abstract syntax (HOAS)

The error-prone “bureaucracy of syntax” exemplified by our flawed `subst` function has been bothering metaprogrammers since the dawn of metaprogramming, and logicians before them. Several techniques have been proposed to alleviate these difficulties.

Higher-order abstract syntax [Church, 1940, Huet and Lang, 1978] is a popular technique which leverages the host language’s own substitution mechanism to implement substitution in the object language.

Figure 1.1b implements HOAS by changing the `Fun` data type — instead of holding a plain name and a body, it holds an actual host-language function from `Exp` to `Exp` — and by removing the `Var` data type. This way, the need for a subst function disappears!

Below is how the `reduce` function could be implemented with the GADT + HOAS abstract syntax tree of Figure 1.1b. Notice that the mistakes mentioned earlier on the plain AST form are no longer possible.

```
def reduce[A]: Exp[A] => Exp[A] = {
  case Lit(v)           => Lit(v)
  case Add(Lit(l), Lit(r)) => Lit(l + r)
  case Add(Lit(v), r)    => Add(Lit(v), reduce(r))
  case Add(l, r)         => Add(reduce(l), r)
  case App(Fun(f), a)    => f(a) // no error-prone subst needed
  // case App(f, a)      => App(reduce(a), f) // mistake is now ill-typed!
  case App(f, a)         => App(reduce(f), a)
  case Fun(f)            => Fun(f)
}
```

However, we should note that a simplistic HOAS approach as presented here is fairly limited. For instance, it could not really be used without adaptations to implement a pretty-printer for the expression trees.

Moreover, HOAS prevents the free deconstruction and transformation of binding structures: In order to inspect the body of a function, one needs to provide an argument to substitute for the occurrences of the variable bound by the function. This argument will typically be an occurrence of the new binding used as a result of the transformation, or a concrete value if we want to remove the binding (inlining the argument). Therefore, we need to know what the final binding structure will look like before we can even look at the body of the function. To see why this is problematic, consider a transformation that inlines a reducible function applications when the function uses its parameter at most once. The problem is that we have no way of knowing how many times the function uses its parameter *before* we have passed an argument to substitute in the body of the function — i.e., before we can make the decision whether to inline the function or not! In a way, this limitation is due to substitution being hard-wired into the definition of HOAS structures, while other slightly more subtle metaprogramming tasks manipulating bindings are not, and thus feel like “second-class” citizens.

Another noteworthy limitation of HOAS is that it delays and duplicates the execution of transformations: every `Fun` AST node stores a closure that is only executed on-demand, and more specifically *every time* the function body needs to be inspected, which re-computes all the transformations applied on that body so far. While this is probably fine for verification approaches mostly concerned with the theoretical soundness of mechanized semantics, it is not realistic for practical metaprogramming applications.

Parametric higher-order abstract syntax (PHOAS)

Yet another problem of the basic HOAS formulation is that while it makes sure that ill-typed terms cannot be represented, it does not prevent the representation of AST values that do not correspond to actual terms of the object language. For instance, in the previous HOAS AST, one can construct the `Exp[Int => Int]` value `Fun((x: Exp[Int]) => if (x == Lit(0)) Lit(1) else x)` does not correspond to any term expressible in the object language.

Chlipala [2008] proposed a *parametric* higher-order abstract syntax approach (PHOAS), which prevents this specific problem and also makes the syntax much easier to use for various analysis and transformation tasks.

However, the two other limitations of HOAS are still present in PHOAS: it delays and duplicates the execution of transformations, and prevents easily inspecting the bodies of functions without either rebuilding a new function (before we get to see the function body) or somewhat compromising on the hygiene guarantees offered by (P)HOAS in the first place (e.g., if we replace variable occurrences by dummies or de Bruijn indices,⁵ which opens us up to the same kind of problems as in the plain AST).

Internal typing

In practical metaprogramming approaches, it is often desirable to keep the AST *internally typed*, meaning that AST nodes should store a runtime representation of the types of the terms they represent, which can be used to influence program analysis and transformation processes down the line.

This can be done by adding parameters in our AST cases, to hold runtime type representation values encoded with a similar GADT as for expressions. Figure 1.1c shows an approach which combines PHOAS with internal typing.⁶ In a declaration of the form `'sealed abstract class Term[T, V](ty: Type[T])'`, the term `ty` is a class constructor parameter — in other words, instances of `Term[T, V]` require a value `ty` of type `Type[T]` in order to be constructed, which will be passed by each subclass of `Term`.

Now the reduce function looks as follows:

⁵It is possible to make de Bruijn indices scope-safe via the type system [Chen and Xi, 2005], but they are still fundamentally low-level “unhygienic” implementation details making it easy to confuse bindings (even within the bounds of scope safety), leading to surprising results [Kiselyov et al., 2016].

⁶For concision, we use the Scala 3 syntax for first-class polymorphic function types, of the form `[X] => T[X]`, which means $\forall X. T[X]$, and can also be encoded in Scala 2 with `abstract class F { def apply[X]: T[X] }`.


```
def reduce[A]: Exp[A] => Exp[A] = e => [V] => e[V] match {
  case Lit(v)           => Lit(v)
  case Var(v, t)         => Var(v, t)
  case Add(Lit(l), Lit(r)) => Lit(l + r)
  case Add(Lit(v), r)     => Add(Lit(v), reduce(r))
  case Add(l, r)          => Add(reduce(l), r)
  case App(Fun(f), a, t)  => f(a)
  case App(f, a, t)       => App(reduce(f), a, t)
  case Fun(f, t, u)       => Fun(f, t, u)
}
```

The curried value-lambda and type-lambda $e \Rightarrow [V] \Rightarrow \dots$ may be a little difficult to read. This is a function expression taking an e parameter (of type $\text{Exp}[A]$) and a V type parameter. In its body, the function applies e to type V — remember that **type** $\text{Exp}[A] = [V] \Rightarrow \text{Term}[T, V]$.

Notice how we need to propagate type representations manually, i.e., one has to pass arguments like t in polymorphic constructors like `Var`. The practice in Scala is usually to make the runtime type representations implicit (using Scala’s implicit parameters feature), so as to somewhat alleviate this burden, but anyone with experience manipulating internally-typed GADT expression trees in Scala knows that it is still very painful and time-consuming (in part because of their interaction with the delicate typing aspects of GADTs).

As we can see, this representation is getting quite a bit “hairy” and more difficult to manipulate, yet what we have seen is still far from the practical metaprogramming and optimization tasks that we are targeting in this thesis. Typically, the design of a DSL and of its domain-specific optimizations quickly becomes entangled with these lower-level AST implementation concerns, which get in the way of the DSL design activity.

Moreover, as the supported type system becomes more elaborate (it is common for Scala DSLs to leverage Scala’s advanced type system features), both the internal and external representations of the corresponding types need to keep up, which can make them much more complicated.

In Section 1.3, we will see that Squid allows us to reap the advantages of an advanced type-safe internally-typed syntax while exposing only a simple quoted interface to the metaprogrammer, and without some of the drawbacks of higher-order approaches.

1.2.2 Existing Scala Quasiquotes

Scala Reflection quasiquotes

In Scala, different forms of quotation coexist. While, for instance, `"2+2"` denotes a string made of characters `'2'`, `'+'`, and `'2'`, when prefixed with `q` as in `q"2+2"` it represents an *abstract syntax*

tree (AST) equivalent to:

```
q" 2+2 " ==  
  Apply(Select(Literal(Constant(2)), TermName("$plus")),  
        List(Literal(Constant(2))))
```

Code quasiquotation using the ‘q’ string prefix was a technique developed by Shabalin et al. [2013] in order to facilitate manipulating the AST constructs of the Scala Reflection API (nick-named *scala-reflect*) while defining Scala macros [Burmako, 2013]. It is available as part of the Scala standard library.

Advantages of a quoted syntax for abstract syntax trees

As we can see in the example above, expressing code using the quoted form `q" 2+2 "` is much more concise than using the “explicit” (non-quoted) form. The explicit form also exposes details of the internal encoding of Scala’s AST which are not usually relevant to metaprogrammers, such as the names of abstract syntax constructs (`Apply`, `Select`, etc.) and the JVM encodings of operator names like `$plus`.

Thus, we can already start to see why a quoted approach has advantages over the explicit approaches described in Section 1.2.1 once the AST to manipulate becomes closer to more featureful real-world programming languages.

Quoted code manipulation

In Scala, quasiquotes can be used as both expressions and patterns. Syntax `${ . . . }` is used to *unquote* terms from inside a quasiquote.⁷ (When the unquoted term is a simple variable, the curly braces can be omitted.) In quasiquote expressions, unquoted terms are *inserted* into the surrounding code. In quasiquote patterns, unquotes *extract* the terms found in their positions, matching them with the unquoted pattern. For example, the following expression:

```
q" 2 + 1 " match {  
  case q"$n + 1 " => q"$n - 1 "  
}
```

evaluates to `q" 2 - 1 "`.

It is straightforward to write a version of the reduce function presented above using Scala quasiquotes:

⁷Unquote [Abelson et al., 1991] is also referred to as *anti-quote* [Mainland, 2007] and *escape* [Taha and Sheard, 2000].

```

def reduce: Tree => Tree = {
  case ConstLit(n)
    => ConstLit(n)
  case Ident(name)
    => Ident(name)
  case q"${ConstLit(n)} + ${ConstLit(m)}"
    => ConstLit(n + m)
  case q"${ConstLit(n)} + $b"
    => q"${ConstLit(n)} + ${reduce(b)}"
  case q"$a + $b"
    => q"${reduce(a)} + $b"
  case q"(($ident: $t0) => $body)($a)"
    => reduce(body.transform { case `ident` => a })
  case q"$f($a)"
    => q"${reduce(f)}($a)"
  case q"($ident: $t0) => $body"
    => q"($ident: $t0) => $body"
}

```

Where `Tree` is the type of Scala AST nodes. We used a custom `ConstLit(_)` constructor synonym to mean `Literal(Constant(_: Int))` for brevity. The expression `t.transform(f)` traverses some tree `t` trying to apply partial function `f` on each of its subterms.

Note that this particular usage of `transform` is not hygienic, notably because it ignores shadowing (we discuss these issues in the next section). Yet, this sort of cavalier applications of code transformation primitives are often seen in Scala macros in the wild, which is part of the reason why Scala macros have had the reputation of being particularly brittle.

1.2.3 Limitations of Scala Reflection Quasiquotes

Despite having achieved widespread adoption, Scala Reflection quasiquotes have important limitations that restrict their potential applications to metaprogramming.

This subsection serves not only as a laundry list of the ways in which Scala Reflection quasiquotes are insufficient — more importantly, it sets the stage for describing the properties and static guarantees that Squid quasiquotes achieve in response to them.

Lack of Static Typing

Scala Reflection quasiquotes fall into the “Lisp-style” analytic category of quasiquotes defined in Section 1.1, since they allow the inspection of program fragments through pattern matching,

but are not statically typed.⁸

More precisely, Scala Reflection quasiquotes only check (at compile-time) for syntactic well-formedness, but do nothing to prevent the expression of ill-typed and ill-scoped code. For instance, the program fragment `q"Math.pow(x, List())"` happily compiles (even in a context without a surrounding binding for `x`) and will generate ill-formed code.

As a stop-gap measure to cope with macros expanding into ill-formed code, the Scala 2 compiler re-type-checks the code resulting from all macro expansions.

This state of affairs is not only less than ideal for performance reasons, but also because it means that macro developers cannot leverage the benefits of static typing when manipulating code, and macro users often end up with compilation errors pointing to invisible macro-generated code which cannot be easily debugged.

Lack of Hygiene

Informally, hygiene is the property that variable bindings do not “get mixed up” as the result of a program transformation; quasiquotes of the Lisp family generally lack such property, though some fare better than others.⁹

The `reduce` transformation we have seen earlier in this section is unhygienic because it is unsound in the presence of shadowing. For example, it will transform `q"((x:Int) => (x:Int) => x)(1)"` into `q"(x:Int) => 1"` instead of transforming it into `q"(x:Int) => x"`. The root cause is that bindings in Scala Reflection quasiquotes are implemented using plain `String` names, which may clash with one another and produce unexpected results.

Hygiene problems manifest in a variety of ways. In the context of macro systems and of Scala macros in particular, two hygiene problems which commonly arise are:

1. Newly-introduced variable bindings clashing with bindings already present in the original program. To work around this problem, one has to manually generate “fresh names” and use them exclusively in generated code (the *gensym* approach of Lisp macros).
2. References to global symbols (such as `math.pow`) in macro-generated code may change meaning if the surrounding code in which the macro expands happens to bind a value with the same name (for instance if it defines a different `math` object with its own `pow` method). To avoid this, macro authors need to fully-qualify the names of the symbols they use (as in `q"_root_.scala.math.pow(...)"` in our example).¹⁰

⁸In that sense, Scala Reflection quasiquotes are very close to the plain simple AST representation of Figure 1.1a in terms of safety properties.

⁹Notably, Template Haskell [Sheard and Jones, 2002] has some mechanisms to automatically create fresh names in generated programs, eschewing many potential metaprogramming errors, although these safety measures can be circumvented easily.

¹⁰The `_root_` package is used to fully qualify identifiers in the global namespace without any possibility of

The creators of Scala Reflection quasiquotes [Shabalin et al., 2013] were well aware that hygiene was an important limitation of their approach. Shabalin [2014] experimented with the idea of adding hygiene on top of Scala Reflection quasiquotes, but his system was fairly complicated and its soundness properties unclear (the formalism came without a proof); consequently, it was never implemented.

We further discuss the problem of hygiene in metaprogramming in Section 1.4.1.

No propagation of internal typing

As explained in the previous section, maintaining internal typing is often desirable in metaprogramming applications. Scala Reflection quasiquotes have the capability of storing internal typing information, but they do not retain or propagate it upon transformation.

For instance, when calling the `reduce` function above on a program `pgrm`, even if `pgrm` is internally annotated with typing information, this information is lost and is not propagated into the transformed program.

Essentially, given two ASTs `a` and `b` both internally assigned type `Int`, the term `q"$a + $b"` will *not* be assigned type `Int`, unless it is type checked again or manually annotated (e.g., as `q"$a + $b".withType(IntType)`, an error-prone approach).

Lack of normalization of syntactic details

Many syntactic details not relevant to the semantics of a program fragment are abstracted after parsing, when the program is put in abstract syntax tree form. For example, `2 * 2 + 1` and `(2 * 2) + 1`, which are not meaningfully different, parse into the same AST.

Yet many more non-semantic details can be abstracted away after name resolution and type checking have been performed, such as: the precise names of bound variables, whether type parameters are explicitly specified (*versus* inferred), the resolution of implicit arguments, syntactic sugar like `for` comprehensions (similar to Haskell `do` expressions), etc.

Because they are untyped, quasiquote systems such as the one provided by Scala Reflection cannot consider equivalent terms as equivalent when they differ only in these superficial ways. For example, pattern `q"Some[Int](42)"` will not match `q"Some(42)"` (and conversely), despite them being equivalent after type checking.

As another example, given some function `f` of type `Int => Int`, the following code fragments are all “semantically” equivalent if we look past name resolution, type inference, and desugaring differences:

```
f(Int.MaxValue)
```

confusion due to name clashes.

```
f.apply(Int.MaxValue)

f(scala.Int.MaxValue)

import Int.{MaxValue => MV}; f(MV)

f(Int.MaxValue): Int

f(Int.MaxValue: Int)
```

Yet, a quasiquote pattern such as `q"$fun(Int.MaxValue)"` will only match the first one (yielding `fun = q"f"`).

This is problematic because it means that when macro writers or DSL designers want to match on certain usage patterns, they have to handle all equivalent representations and their possible combinations, making the task unfeasible.

In practice, Scala macro writers try to only analyse fully-typed expressions for this reason. When the Scala compiler type checks a program, it rewrites all expressions into their “fully-explicit” form — in the example above, all forms except the last two are rewritten into `f.apply(scala.Int.MaxValue)`.

However, relying on the assumption that terms are in type-checked form is problematic, as any subsequent transformations may violate that assumption. Moreover, there is no way of checking that expression and pattern quasiquotes are always written in that form, so it is extremely easy to introduce subtle code transformation problems by deviating from this implicit normal form.

1.3 Code Manipulation with Squid Quasiquotes

In this section, I introduce Squid’s approach to analytic metaprogramming through statically-typed and hygienic quasiquotation, explaining the wide array of metaprogramming features that it supports.

1.3.1 Basics of Squid Quasiquotes

Squid quasiquotes are prefixed with the ‘`code`’ identifier, and manipulate AST nodes of type `Code[T]`, where `T` reflects the type of the represented object term (like in the GADT approaches of Figure 1.1). For example, `code"42.toDouble"` has type `Code[Double]`.

Probably the main difference with Scala Reflection quasiquotes is that Squid type checks the quoted code fragments while they are compiled, and uses the resulting typing information to create the appropriate AST nodes. As a result, the AST nodes are always internally represented in a fully-typed form: all type parameters are specified, the code is desugared (e.g., `f(123)` is represented as `f.apply(123)`) and implicit arguments are inferred. This is the case even

when the quasiquote itself does not mention type parameters, uses syntax sugar, and/or omits implicit arguments.

For example,¹¹ the quasiquote `code"List(1,2).map(_ + 1)"` is *equivalent* to:

```
code"scala.Predef.List.apply[Int](1, 2)
    .map[Int, List[Int]]((x: Int) => x + 1)(List.canBuildFrom[Int])"
```

Indeed, both of these forms will compare equal by the standard `==` equality operator.

Under the hood, Squid quasiquotes are macros that produce the boilerplate necessary for constructing or deconstructing AST nodes corresponding to the code being quoted.

1.3.2 Pattern Matching and Rewriting

Just like Scala quasiquotes, Squid quasiquotes support pattern-matching. However, type annotations are often required to help with Scala's local type inference. For example, the pattern `code"$x + 1"` does not type check, as the Scala type checker cannot not know which '+' method is implied when the type of `x` is unknown. The pattern matching example in Section 1.2.2 is now written:

```
code"2 + 1" match {
  case code"($n: Int) + 1" => code"$n - 1"
}
```

To help define concise rewriting transformations, Squid provides a convenience `rewrite` method which traverses a program in top-down or bottom-up order (which can be configured) and applies a transformation to each of its sub-terms, while checking at compile-time (of the metaprogram) that the transformation is type-preserving.

1.3.3 Type Representation Implicits

In order to satisfy the requirement that program fragments be internally typed (i.e., they should contain runtime information about the types of the terms that they encode), Squid requires functions manipulating code which contains statically-unknown types to pass along associated runtime type representations for these unknown types.

As hinted in Section 1.2.1, the most convenient way to do so in Scala is via implicit parameters. Squid defines the `CodeType` type class for this purpose. As an example, the following function returns an empty option term for any type `T`:

```
def foo[T: CodeType] = code"Option.empty[T]"
```

¹¹In Scala, `map` takes an implicit `CanBuildFrom` parameter used to allow mapping over heterogeneous types [Odersky and Moors, 2009], but whose precise semantics is irrelevant to this presentation.

Chapter 1. Statically-Typed Code Manipulation with Analytic Quasiquotes

The syntax `T: CodeType` in type parameter lists is shorthand for including a corresponding implicit parameter of type `CodeType[T]`, so the code above is equivalent to:

```
def foo[T](implicit _: CodeType[T]) = code"Option.empty[T]"
```

When `foo` is called as e.g., `foo[Int]`, an implicit type representation, of type `CodeType[Int]`, is synthesized and passed along with the function call, so that the resulting term is the expected `code"Option.empty[Int]"`.

We could also write the call to `foo[Int]` by passing the implicit argument explicitly,¹² using the `Squid codeTypeOf` primitive to request a type representation for `Int`:

```
val intType = codeTypeOf[Int]
val res = foo[Int](intType)
```

Type representation values can also be obtained from terms of the corresponding type, so we can write:

```
assert(res.T == codeTypeOf[Option[Int]])
```

where `==` is used to compute type equivalence.

1.3.4 Matching and Extracting Unknown Types

To define type-parametric rewrite rules, `Squid quasiquotes` allows the extraction of *types* (not just *terms*) from quoted patterns.

In the example below, given some `pgrm` fragment, we transform calls to `foldLeft` on `List` objects into imperative `foreach` loops:

```
def lower[T](pgrm: Code[T]) = pgrm rewrite {
  case code"($ls: List[$t]).foldLeft[$r]($init)($f)" =>
    code" " "
      var cur = $init
      var xs = $ls
      while (xs.nonEmpty) {
        val x = xs.head
        xs = xs.tail
        cur = $f(cur, x)
      }
      cur
    " " "
}
```

¹²In practice, passing type representations around explicitly is almost never necessary.

which can be used as follows, for instance:

```
lower(List(1, 2, 3).foldLeft(0)((acc, x) => acc + x).toDouble)
```

and returns (a desugared version of) the following program fragment:¹³

```
code" " "
{
  var cur = 0
  var xs = List(1, 2, 3)
  while (xs.nonEmpty) {
    val x = xs.head
    xs = xs.tail
    cur = cur + x
  }
  cur
}.toDouble
" " "
```

Note that in Scala, multi-line quotations are introduced with triple quotation marks `" " "` and that the “operator syntax” `p rewrite f` is equivalent to the “method syntax” `p.rewrite(f)`.

Extracted type representations are values and *not* types

Any type extracted from a quasiquote pattern such as `case code" ...[$t0] ... "` results in a *value* `t0` of type `CodeType[T0]`, where `T0` is an existential type which *cannot be directly referred to* in the right-hand side of the pattern matching case.

Indeed, only values can be extracted from the Scala “string interpolator” syntax which Squid relies on for its `code` quasiquote patterns. So `t0` has to refer to the runtime type representation for `T0`, and not `T0` directly. Thankfully, we can access `T0` indirectly through the path-dependent type `t0.T` — this works because every `CodeType[A]` value `ty` contains a type member named `T` which reflects its type argument `A`, so that `ty.T` refers to `A`.

For example, one can write:

```
def bar(x: Code[Any]): Code[Any] = x match {

  case code" Some[$t0]($_) " =>
    val c = foo[t0.T]
    // or equivalently:
    val c = foo(t0)
```

¹³Notice that the β reducible expression `((acc, x) => acc + x)(cur, x)` was automatically reduced by Squid as it is equivalent to a trivial let binding.

```
assert(c == code"Option.empty[$t0]")

case _ => x
}
```

Above, the call to `foo[t0.T]` is elaborated into `foo[t0.T](t0)` by the compiler — the type representation value `t0` is picked up from the current scope automatically, based on its static type, and passed as the value for `foo`'s implicit type representation parameter.

Bounding extracted types

It is sometimes necessary to constrain pattern type variables to conform to some specific subtyping bounds.

In particular, this is sometimes needed to make a pattern even type check. For instance, consider the signature of the `orElse` method¹⁴ defined on the `Option[T]` type in Scala:

```
Option[A] <: {
  def orElse[B >: A](alternative: => Option[B]): Option[B]
  ...
}
```

Notice that type parameter `B` is lower-bounded by `A`. When trying to write an optimization for `orElse` in Squid quasiquotes, the following rewrite rule will fail to compile:

```
case code"Some[$ta]($lhs).orElse[$tb]($rhs)"
=> lhs
```

because extracted type `tb` is not known to be a supertype of `ta`, and therefore the call to `orElse` in the pattern does not type check. To fix this, Squid provides a lightweight syntax to require some bounds on extracted types:

```
case code"Some[$ta]($lhs).orElse[$tb where (ta <:< tb)]($rhs)"
=> lhs
```

The refined version above compiles and has the expected semantics.

Bounds on extracted types are checked at runtime, even though a runtime check is not always necessary (like in the example above, where the check should succeed by construction, since `orElse` cannot be called with non-conforming type arguments).

¹⁴`orElse` is used to fall back to an alternative option if the first is `None`. Expression `Some(x).orElse(y)` returns `Some(x)` and `None.orElse(y)` returns `y`.

1.3.5 Nonlinear Pattern Variables

Squid allows using the same pattern variable several times in a single pattern. For instance, the following rewrite rule simplifies an `if` expression with the same subexpression (up to alpha equivalence) on its two branches:

```
case code"if ($cond) $cde else cde"
=> code"$cond; $cde" // re-insert cond in case it has side effect
```

Due to a Scala restriction in the syntax of patterns, only one of the multiple occurrences of the pattern variable is marked with a dollar sign.

As another example, the following pattern matches a call to `map` and `take` where the input and output types of `map` match, and rewrites it into a call to the `mapFirst` function (a hypothetical `map` implementation which maps only the first `n` elements and therefore does not change the overall list's element type):

```
case code"(xs: List[$ta]).map(f: ta => ta).take($n)"
=> code"xs.mapFirst($n, $f).take($n)"
```

1.3.6 Cross-Quotation References

An important feature of a flexible quasiquotation system is the ability to manipulate open terms [Taha and Sheard, 2000].¹⁵ Since Squid quasiquotes are type-checked and hygienic, a program fragment like `code"x + 1"` is not valid on its own, as `x` is not defined — contrast this with current Scala quasiquotes, where `q"x + 1"` is perfectly valid even if `x` never ends up being bound in the generated program.

However, within a context where `x` is bound at *the same quotation depth*¹⁶ as its reference, `code"x + 1"` becomes a valid expression. For instance, `code"(x: Int) => ${bar(code"x + 1")}"` is a valid quasiquote (of type `Code[Int => Int]`): when evaluated, the inner quasiquote embeds a reference to the outer `x`, returning a program fragment to be processed by `bar`, the result of which (which will presumably contain references to `x`) will be inserted into the outer quote, which binds `x`.

1.3.7 Cross-Stage References and Cross-Stage Persistence

Referring to variables defined across different quotation depths is normally forbidden. For instance, the following program is rejected by Squid with a compile-time type error:

¹⁵A capability notably missing from the earlier Scala Reflection statically-typed quotation approach, the *reify* macro (see Section 1.7).

¹⁶The notion of quotation depth used here is purely syntactic: it corresponds to the number of surrounding quotes minus the number of surrounding unquotes. Why this simple notion is sufficient to guarantee scope-safety and hygiene is explained later on in the thesis.

```
val x = Console.readInt
code"println(x + 1)"
```

The given error offers a suggestion: “Perhaps you intended to use a cross-stage reference, which needs the `@squid.lib.crossStage` annotation.” This error is reported by Squid in the first place because situations where cross-stage references occur very often indicate a programmer error. However, when a cross-stage reference is actually desired, programmers can use the `@crossStage` annotation as follows:¹⁷

```
import squid.lib.crossStage
@crossStage val x = Console.readInt
code"println(x + 1)"
```

Squid will interpret this code as the construction of a program fragment which contains a reference to a value living at the *current stage* (the stage during which program fragments are manipulated). We call such references ***cross-stage references***. Moreover, built-in support for cross-stage references is called ***cross-stage persistence***.

Note that in the example above, `println` is *not* a cross-stage reference, because it is a globally-accessible symbol (it desugars to the `scala.Predef.println` selection path). Similarly, the `+` method in `x + 1` is accessible from the global `Int` type, so it is statically resolved and does not incur any cross-stage persistence.

Program fragments containing cross-stage references typically cannot be interpreted outside of the current runtime environment, because they would lose access to the corresponding current-stage values. For instance, trying to dump a generated program containing cross-stage references into a source file will normally not succeed, unless the corresponding current-stage values happen to be serializable, in which case Squid will generate the corresponding deserialization code.

The main use of cross-stage persistence is for retaining cross-stage references in program fragments which are eventually runtime-compiled, a process which we describe next.

1.3.8 Runtime Compilation and Cross-Stage Persistence

After composing a program at run time using quasiquotes, one can then either dump a stringified version of the code inside a file to be compiled and run later, or runtime-compile it on the fly, using the `.compile` method, which produces bytecode that can then be run efficiently.

After the one-off cost of runtime compilation is amortized, a runtime-compiled function definition such as `val f = code"(x: Int) => x + 1".compile` will be as efficient to call as if it had been defined with `val f = (x: Int) => x + 1`.

¹⁷Alternatively, Squid supports writing `code"println(%(x) + 1)"` to indicate that `x` is a cross-stage reference.

Going back to cross-stage persistence, the following example code passes a current-stage mutable array into a program fragment which is then runtime-compiled. Subsequently mutating the array shows that the cross-stage reference has indeed been preserved, and still refers to the same current-stage array instance:

```
@crossStage val arr = Array(0, 0, 0)
val test = code"(i: Int) => arr(i)".compile
println(test(0)) // prints 0
arr(0) = 1
println(test(0)) // prints 1
```

1.3.9 Automatic Function Lifting and Unlifting

Squid supports a convenience feature called *automatic function lifting*: upon insertion, Squid automatically lifts any host-language function, of type `Code[A] => Code[B]`, into an object language function, of type `Code[A => B]`, and immediately inlines it, when possible.

As an example, the following code evaluates to `code"(x: Int) => x + 1"`:

```
val f = (y: Code[Int]) => code"$y + 1"
code"(x: Int) => $f(x)"
// or equivalently:
code"$f"
```

In essence, automatic function lifting has the same semantics as desugaring `$f(a)` into `${f(code"a")}` when `a` is pure expression (for some notion of purity — see Section 4.4.2 for details), and into `val x = a; ${f(code"x")}` otherwise.

The reverse transformation, from `Code[A => B]` into `Code[A] => Code[B]`, known as *automatic function unlifting*, is also provided by Squid as an implicit conversion.

1.3.10 Higher-Order Pattern Variables

Squid provides a very simple form of higher-order matching [Pfenning and Elliott, 1988, de Moor and Sittampalam, 2001] which directly mirrors automatic function lifting.

A pattern like `case code"(x: Int) => $body: Int"` will not match a lambda where `body` makes use of `x`, while the following pattern will: `code"(x: Int) => $f(x): Int"`, giving to the extracted pattern variable `f` type `Code[Int] => Code[Int]`. Applying `f` to some `Code[Int]` value will replace all usages that `f` made of `x` in the original program fragment.

Note that this effectively reintroduces some of the limitations of HOAS criticized earlier in this chapter. We will see in later chapters of the thesis how to use a more fine-grained and safer approach to manipulating bindings and open code, removing these limitations.

Higher-order pattern variables in quasiquote-based pattern matching were suggested before us by Sheard et al. [1999], but we do not know of any actual implementation of the idea, beside ours.

1.3.11 Code Combinators

Quotations are not always the most appropriate way of representing the possible shapes of program fragments — despite their resolutely semantic flavor (they are more “semantic” than “syntactic” since they ignore most syntactic details), sometimes Squid quasiquotes are still too concrete.

For this reason, Squid provides a number of “code combinators” [Chen and Xi, 2005, Kameyama et al., 2015, Kiselyov et al., 2016] which allow the construction and deconstruction of program fragments in a more generic way.

The most common combinator, `Const`, is the constructor for constant values. For instance, `Const(27)` is the same as `code"27"`. This combinator is needed because the syntax of quotations does not allow distinguishing between syntactic categories like constants versus expressions — i.e., one cannot insert a plain `Int` as if it were a code value (unless an implicit conversion is used, but we try to avoid those, as they tend to make metaprograms unnecessarily confusing and slow to compile [Rompf, 2016]).

As another example, `LeafCode()` is a custom extractor (a Scala construct which can be used in patterns) defined by Squid to match any expression that has no sub-terms, such as variable references, constants, and global symbols like `scala.math`.

There are several other code combinators provided by Squid, such as `MethodApplication` — which will be used and explained further in Section 1.5.

1.3.12 Call-By-Name Reduction Example

Using Squid, we can now rewrite the `reduce` example seen in Section 1.2.2, but in a type-safe and hygienic way as shown in Figure 1.2.

`VariableRef()` is an extractor defined by Squid that matches variable references, but does not extract any useful information from them.

This `reduce` implementation is not far, in terms of concision, from the one we have seen in the previous section, but it does much more: notably, it statically makes sure that all quoted expressions are well-typed, it manipulates bindings in a hygienic way, and it propagates internal typing information automatically.

```

1  def reduce[T: CodeType]: Code[T] => Code[T] = {
2    case Const(n)
3      => Const(n)
4    case x @ VariableRef()
5      => x
6    case code"(${Const(a)}: Int) + (${Const(b)}: Int) "
7      => Const(a + b)
8    case code"(${Const(a)}: Int) + ($b: Int) "
9      => code"${Const(a)} + ${reduce(b)} : T"
10   case code"($a:Int) + ($b: Int) "
11     => code"${reduce(a)} + $b : T"
12   case code"((p: $t0) => $body(p): T)($a) "
13     => body(a)
14   case code"(f: $t0 => T)($a) "
15     => code"${reduce(f)}($a) "
16   case e @ code"(p: $t0) => $body(p) "
17     => e
18 }

```

Figure 1.2 – The reduce function implemented with Squid quasiquotes.

Extracted types and GADT reasoning

Notice the `: T` type ascriptions¹⁸ on lines 9 and 11. They are necessary to make the program type check. Without such ascriptions, the unannotated term `code"${reduce(a)} + $b"` would be inferred by Squid to have type `Code[Int]` instead of the expected `Code[T]`, and the Scala compiler would complain about a type mismatch (*expected*: `Code[Int]`; *found*: `Code[T]`).

As we will further discuss in Section 1.4.4, in this particular branch of the pattern match, since we know that we have matched a term of type `Code[T]` with a pattern of type `Code[Int]`, we can conclude that `T` is a supertype of `Int`, and we *should* be able to upcast `Code[Int]` to `Code[T]`.

However, the Scala type checker has no specific knowledge of Squid pattern quasiquotes, and so it has no way to know that in this particular pattern branch, `T` is related to `Int`.¹⁹ Fortunately, Squid keeps track of such uncovered type relations, and it provides a convenience implicit conversion called `coerce`, which is used to convert values between known-to-be-related types. To do this, the Squid pattern macro compares the type of the quoted pattern with the type of the scrutinee, recursing into the subcomponents of each type while following the variance of the corresponding type constructor positions, and recording a list of discovered subtyping

¹⁸In this example, notice that `T` refers to a *type*, not a *type representation*, so it should not appear unquoted.

¹⁹This is a technical limitation, related both to the way Scala 2 pattern macros are expanded, and to the lacking support for GADT reasoning in Scala 2 (see Appendix A for more details and explanations).

relations.

This `coerce` function is applied implicitly at line 7 thanks to the expected type of `reduce`, but expected types do not propagate into quotes (a fundamental limitation of Squid's macro-based approach), so lines 9 and 11 need some help. We annotate the expressions inside the quote with `: T` so that the `coerce` function is invoked implicitly while type-checking the expression (the corresponding synthetic call is then discarded by the quasiquote macro, so it does not become part of the resulting program fragment).

1.4 Safety Properties of Squid Quasiquotes

In this section, we briefly review Squid's safety properties and how they are achieved, including hygiene, scope safety, and type safety. This is only an informal overview; a safer version of Squid (referred to as *Contextual Squid*) is formalized and proven safe later, in Chapter 6.

1.4.1 Hygiene

Hygiene problems were historically solved in macro systems such as Scheme by treating bindings from the original program and bindings introduced by a macro expansion differently, and by preventing one kind from mixing up with the other [Kohlbecker et al., 1986].

Unfortunately, this does not generalize to program transformations commonly found in compiler passes, which cannot always be expressed in terms of simple macro expansion. Even in a language like Scheme, quasiquotes are still fundamentally unhygienic — as a matter of fact, the high-level facilities provided by Scheme for writing hygienic macros are not based on quasiquotation, but on more limited and less expressive code manipulation primitives.

As a good example of the tricky nature of hygiene, while developing Squid I discovered a hygiene bug in the Scala compiler²⁰ that had been left undetected for many years, and that was spanning several major versions of the compiler. The code:

```
val a = 100; ({ val a = 0; (c: Int) => c })(a)
```

used to be transformed into:

```
val a: Int = 100; { val a: Int = 0; ((c: Int) => c).apply(a) }
```

during type checking, altering the semantics of the expression. Squid would have rejected the expression of such an unsound rewriting at compile time of the program transformer.

Squid expression and pattern quasiquotes are statically type-checked and put into a normal form where symbols are uniquely identified, so that for example `code"Math.pow($x, 2)"` and `{import Math.pow; code"pow($x, 2)" }` are equivalent. Moreover, bound variables in the source

²⁰<https://issues.scala-lang.org/browse/SI-10170>

program cannot interfere with variables introduced by transformers, because Squid decides statically when and where variable capture occurs and uses fresh variable identifiers for each binding, preventing unbound references and unexpected captures.

1.4.2 Scope Safety

The handling of binding decomposition through higher-order patterns variables (see Section 1.3.10) gives Squid some similar properties as the higher-order abstract syntax approaches studied in Section 1.2.1.

Squid's approach is as scope-safe as PHOAS. But in contrast to PHOAS, it is much simpler (not needing to encode terms as first-class polymorphic functions) and in contrast to (P)HOAS, it executes term construction and transformations eagerly, caching the result — i.e., when a binding is created by a quote, its body is immediately reified, as opposed to the (P)HOAS approach where the body remains suspended inside a closure of the `Fun` constructor.

However, Squid does suffer from the other major limitation of higher-order approaches: its handling of bindings is limiting and cannot express more advanced binding manipulations use cases. For example, in Chapter 5, we find a transformation that requires the use of an unsafe escape hatch from Squid's HOAS-like system. We will see how to remove the need for this escape hatch in Chapters 6 and 7, where we see how to make the handling of open terms much more flexible (dropping the requirement for *lexical* scoping), while still ensuring that it is scope-safe and hygienic.

1.4.3 Type Safety

Squid relies on the Scala type checker to check and elaborate all quoted expressions. Therefore, it benefits from a battle-tested implementation which always agrees with the type system of the host language, an important property since we reflect the types of program fragments in their static host-language types.

Moreover, Scala's support for bounded type abstraction and path-dependent types is instrumental in making the type extraction and type-parametric rewriting use cases (explained in Section 1.3.4) work well and soundly within the rest of the type system.

1.4.4 GADT Reasoning

When pattern matching on code values, one usually does not know all the types that may hide inside the subparts of the analysed program. So by inspecting code with pattern matching, one may discover the existence of unknown types, which have to be treated as existential types. Moreover, as we have seen in Section 1.3.12, some relations between known types may be *discovered* when a particular pattern matches. Unsurprisingly, the tricky typing considerations

which arise from this turn out to be precisely similar to the ones that arise from pattern-matching on GADTs.

As mentioned before, Squid palliates some of the limitations of the Scala 2 compiler with respect to so-called GADT reasoning by recording its own type refinement information and applying its own GADT reasoning. This was implemented based on a new understanding of GADTs in a language with subtyping like Scala, a subject which is discussed in some more depth in Appendix A.

1.4.5 Pattern Matching Exhaustiveness

Pattern matching is never checked for exhaustiveness in Squid, which is a known limitation.

In order to exhaustively transform a program using Squid by recursive pattern matching, one should make sure to use a mix of quasiquotes and code combinators handling all of the basic constructs of the language: variable and module references, method applications, lambda abstractions, etc. Surprisingly, there are not so many features to handle in Squid’s default intermediate representation,²¹ since many more advanced Scala features are internally represented in terms of more primitive features like method application (see Section 3.4).

It is interesting to note that in the majority of the metaprogramming use cases I have considered during my PhD, exhaustive pattern matching on all possible constructs of the language was never really needed, except in a quoted implementation of pretty-printing — instead, the bulk of the pattern matching was performed in rewritings or in pattern matching expressions with default cases.

1.4.6 Safety of Rewriting

Implementing term rewriting correctly in the presence of internal typing and subtyping can be surprisingly difficult, as the problem is more subtle than it looks.

For instance, a naive approach to applying a rewriting with pattern type τ and right-hand side type υ on all the subterms of some term t , one could imagine performing the following steps:

- statically check that $\upsilon <: \tau$, to ensure that the rewriting is type-preserving;
- for each subterm s of t , check whether $s.\text{type} <: \tau$; if so, match s with the rewriting’s pattern, and if there is a match, apply the rewriting.

However, the approach outlined above does not work. To see why, consider the case where $\tau = \upsilon = \text{Any}$ (the top-type of Scala’s subtyping lattice) and where the rewriting is of the form

²¹Squid supports various underlying intermediate representations, as explained later in Chapter 3. The default intermediate representation is based on a simple AST structure.

{ **case** Const(_: Any) => **code**"false: Any" }. Using the algorithm above, we would rewrite a term such as `2 + 2` into `false + false`, which obviously does not type check.

To correctly perform rewriting, one needs to make sure, for each subterm *s*, not only that *s.type* <: *T*, but also that *U* should be a subtype of *the type at which the subterm s is used* in the original program. Such considerations are naturally complicated by features like type-parametric rewritings.

Chapter 6 formalizes the general rewriting algorithm (but omits type-parametric rewriting, for the sake of simplicity) and proves its correctness.

1.5 Example: A Quoted ANF Conversion

Correctly handling bindings is one of the most common pitfalls in program manipulation. The higher-order pattern variable (HOPV) technique presented in Section 1.3.10, which is used to match binding constructs, can seem limiting because it extracts functions instead of directly-inspectable terms. In this section, we show that HOPVs are still flexible enough to express some interesting transformations.

Intermediate representations (IR) may automatically normalize terms into forms such as SSA, CPS or ANF, *by construction*. I call these *internally-normalizing* representations. In this chapter, we focus on simple AST representations of programs which are not internally normalizing. However, we will see in Chapter 3 that Squid quasiquotes can be made to work on internally-normalizing representations, thanks to Squid's modular design.

When the IR is not internally-normalizing, it is still possible to perform ANF *conversion* as a type-safe, hygienic transformation expressed with quasiquotes. Figure 1.3 presents such a transformation for our toy lambda calculus with integers and addition, now extended with if-then-else. Note that *rec* is the name of the helper function, and not a keyword of the language.²²

As explained in Section 1.3.9, Squid provides implicit conversions to go back and forth between lifted (`Code[A => B]`) and unlifted (`Code[A] => Code[B]`) function types. Using this facility, variable *k* on line 20 is lifted in order to be inserted, and value `code"join"` on line 26 is unlifted in order to be passed to *rec*.

As an example of execution, the program:

```
val foobar = {
  val foo = 123
  val bar = 42
  (if (true) foo else foo + 2) + bar
}
```

²²I sincerely apologize to all OCaml developers trying to read code using this unsettling naming convention!

```

1  def toANF[T: CodeType](trm: Code[T]) =
2    rec(trm)(identity)
3
4  def rec[T: CodeType, R: CodeType]
5    (trm: Code[T])(k: Code[T] => Code[R]): Code[R] =
6    trm match {
7      case Const(_)
8        => code"val c = $trm; $k(c)"
9      case code"val x: $tx = $a; $body(x)"
10       => rec(a)(x => rec(body(x))(k))
11      case code"($a:Int) + ($b:Int)"
12       => rec(a)(a => rec(b)(b =>
13         code"val add: T = $a + $b; $k(add)"))
14      case code"($f: $t0 => T)($a)"
15       => rec(f)(f => rec(a)(a =>
16         code"val app: T = $f($a); $k(app)"))
17      case code"(p: $t0) => ($body(p): $t1)"
18       => code" " "
19         val f: T = (p: $t0) => ${toANF(body(code"p"))}
20         $k(f)
21         " " "
22      case code"if ($cnd) $thn else $els"
23       => rec(cnd)(cnd =>
24         code" " "
25           val join = $k
26           if ($cnd) ${ rec(thn)(code"join" ) }
27           else      ${ rec(els)(code"join" ) }
28           " " ")
29      case _ => k(trm)
30    }

```

Figure 1.3 – Type-safe, hygienic ANF conversion.

foobar + 1

is transformed by the algorithm of Figure 1.3 into:

```

val c_0 = 123
val c_1 = 42
val c_2 = true
val join_7 = ((lifted_3: scala.Int) => {
  val add_4 = lifted_3.+(c_1)
  val c_5 = 1

```

```
    val add_6 = add_4.+(c_5)
    add_6
  })
  if (c_2)
    join_7(c_0)
  else {
    val c_8 = 2
    val add_9 = c_0.+(c_8)
    join_7(add_9)
  }
```

Notice the hygienic generation of fresh names by Squid in the end program (e.g., `add_4` and `add_6`).

We can generalize our approach to handling other language constructs in a straightforward way. More interestingly, generalizing the approach to handling *arbitrary* computations²³ can also be done by replacing the cases for integer addition and function application with `case MethodApplication(ma)`, which is a helper extractor defined by Squid. This extracts an object `ma` capable of representing any method application, which can then be rebuilt by applying a type-preserving transformation on each of its arguments, as follows:

```
case MethodApplication(ma) =>
  ma.rebuildCPS([T: CodeType, R: CodeType] => rec[T, R])(r =>
    code"val tmp = $r; $k(tmp)")
```

The `rebuildCPS` method takes a polymorphic function (again, here using Scala 3 syntax for polymorphic lambdas) to be applied to each argument passed in the matched method application. This function is in continuation-passing style, which usefully gives it the same shape as `rec` itself, and which is necessary to introduce variable bindings along with each transformed argument. Method `rebuildCPS` also takes a continuation argument that we use to bind the result of the method application to a `tmp` variable.

It is interesting to compare our implementation of A-Normalization to the original Scheme algorithm by [Flanagan et al., 1993]. The continuation-based structure is essentially the same, and the size of the program (19 lines of code in their case) is in the same ballpark. However, our version has the additional advantage to be type-safe and to propagate internal typing (which they do not). Moreover, our version is hygienic by construction, while they need to use the error-prone “gensym discipline” of manually generating fresh names to avoid introducing name clashes. In our case, Squid takes care of these low-level details automatically.

²³Most Scala computational forms are represented as method calls by Squid’s default intermediate representation, as explained in Section 3.4.

```
@macroDef
def naivePower(base: Double, exp: Int): Double =
  // in this scope, base: Code[Double] and exp: Code[Int]
  exp match {
    case Const(exp) =>
      var cur = code"1.0"
      for (i <- 1 to exp) cur = code"$cur * $base"
      cur
    case _ =>
      code"Math.pow($base, $exp.toDouble)"
  }
```

Figure 1.4 – Naive version of the power macro.

1.6 Type-Safe & Hygienic Macros for Scala

In this section, we briefly describe another feature of Squid, which acts like an alternative to the current Scala macros. As a motivating example, consider the typical `power(x, n)` function that raises number `x` to the n^{th} power. We want to write a power macro which expands into a series of multiplications when the exponent argument received is a known constant.

A first version is shown in Figure 1.4. The `macroDef` annotation transforms a method definition into a macro. Like in Scalameta [Burmako, 2017b], the effect of that annotation is that *within the body* of the annotated function, each parameter declared as `p: T` is reinterpreted as a piece of code with type `p: Code[T]` which can be inspected using pattern matching and rewriting.

The macro in Figure 1.4 is “naive” in that it will duplicate the base code, resulting in potentially unnecessary computations and even in changes in program semantics — indeed, program `naivePower(readInt, 2)` will expand into `1.0 * readInt * readInt`. To correct this flaw, we have to first assign the value of `base` to a temporary variable, and duplicate a references to that variable instead. The corrected macro, which binds `base` to an intermediate variable, is presented in Figure 1.5.

1.7 Related Work

We now review some related work.

1.7.1 Existing Quasiquotation Systems

Lisp-style. While the idea of quasiquotation is old [Quine, 1940], Lisp was the language that pioneered its usage as a metaprogramming construct [Bawden, 1999]. Lisp treating code as “just” data meant that no special restrictions or mechanisms were in place to prevent

```

@macroDef
def power(base: Double, exp: Int): Double =
  exp match {
    case Const(exp) =>
      code"val b = $base; ${
        var cur = code"1.0"
        for (i <- 1 to exp) cur = code"$cur * b"
        cur
      }"
    case _ =>
      code"Math.pow($base, $exp.toDouble)"
  }

```

Figure 1.5 – Correct definition of the power macro.

common errors associated with code manipulation, such as unintended variable capture (lack of hygiene), scope extrusion and type mismatches (lack of static typing). Scheme introduced facilities to write *hygienic macros* [Kohlbecker et al., 1986, Abelson et al., 1991, Culpepper and Felleisen, 2004] using safer constructs which separate identifiers appearing at different macro expansion phases (e.g., distinguishing identifiers introduced by a macro from those present in the original program). However, these constructs are much more restrictive than quasiquotation, reducing the expressiveness of this new macro implementation mechanism. Code quasiquotation in Scheme remains unhygienic, so it is mostly used as a lower-level building block. Rhiger [2012b] proposed a finer-grained hygiene system for Scheme-like code quasiquotes, but it does not support pattern matching on code values.

MetaML-style. The idea of code quasiquotation was picked up in a statically-typed context by Taha and Sheard [2000] with MetaML (and subsequently MetaOCaml [Taha, 2004]) to enable multi-stage programming (MSP). The approach was ported to compile-time macros by Ganz et al. [2001] with MacroML. In these systems, quasiquotes can only *generate* and not *inspect* code — though MacroML has some limited form of pattern–template expansion that borrows from Scheme’s hygienic macro system. This style of quasiquotation is provided as an extension of the host language’s type system, which provides static guarantees about the code generated by quasiquotes: it is well-typed and well-scoped, except in the presence of imperative effects, which can lead to scope extrusion (cf. Section 1.1.3).

Template Haskell. With Template Haskell (TH), Sheard and Jones [2002] introduced compile-time metaprogramming to Haskell using quasiquotes which had some notions of type awareness and hygiene, but could easily generate ill-typed and ill-scoped code, therefore providing weaker guarantees than MetaOCaml. Typed Template Haskell (TTH) later added type-safe quasiquotes similar to MetaOCaml. Neither MetaOCaml nor TH/TTH support term deconstruction via quasiquotes in pattern matching.

Quasiquotation in Haskell, Scala, and Squid. A *general* quasiquotation syntax (not restricted to *code* quasiquotes) was introduced in Haskell by Mainland [2007]. Since this new quasiquotation syntax supports patterns, it could in principle be used to implement quasiquote-based code pattern matching in Haskell, but this has never been done (to the best of my knowledge). A similar general quasiquotation system exists in Scala (but is confusingly referred to as a “string interpolator” syntax) and is used by the Scala Reflection API to provide Lisp-like untyped code quasiquotes with support for pattern matching [Shabalin et al., 2013]. This is the very system used by Squid, with the difference that Squid uses type-aware macros in order to extend Scala’s type system (similarly to how MetaML-style quasiquotation extends ML’s type system), enabling static type checking. The Scala reflection API has an alternative type-safe and hygienic *reify*/*splice* system that can be used for program generation (*reify* acts like quotation and *splice* like antiquotation), but that system does not allow the expression of open code and does not support pattern matching, greatly limiting its usefulness — for instance, while it can be used to implement the flawed macro in Figure 1.4, it cannot be used to implement the correct version in Figure 1.5.

Other languages. Several other languages such as F# [Syme, 2006] support different flavors of quasiquotes that fall into the categories defined above. Table 1.1 summarizes the features supported by quasiquotes in Squid and in several other systems. The Stratego snippet uses an example object language, but Stratego is not tied to any particular language. The asterisk (*) on the “well-scoped” and “hygienic” criteria for TTH denotes that these properties are achieved by forbidding any effects in the code generator, which can be restrictive and prevents, e.g., effectful *let*-insertion [Kameyama et al., 2015].

1.7.2 Unification of Runtime and Compile-time Metaprogramming

Squid provides the same type-safe quasiquote-based interface to manipulate code at run time and at compile time. At run time, Squid is typically used for the purpose of runtime compilation (the usual practice in traditional multi-stage programming), or for dumping the resulting code in source files to be compiled later. At compile time, Squid can be used to implement macros in a type-safe and hygienic way (as we saw in Section 1.6). The underlying infrastructure Squid uses to implement these features relies heavily on the incredible work done by Burmako [2013, 2017a] to make the Scala Reflection API usable both at run time and at compile time.

Stucki et al. [2018] later implemented a Squid-style quasiquote API for the new Dotty compiler, providing similar dual run time and compile time capabilities. They introduced a level-counting “Phase Consistency Principle” which corresponds to the usual notion of quotation depth used in MetaOCaml and Squid to define valid cross-quotation references (presented in Section 1.3.6). On the other hand, their approach does not support any cross-stage persistence, unlike Squid (as shown in Section 1.3.7). It should be noted that because of limitations with Scala macros (explained later in Section 3.9.2), Squid originally did not fully support cross-

quotation references, and instead relied on a distinct notion of non-lexically-scoped free variables which we develop in Chapter 6.

1.7.3 Type-Safe Code Manipulation

Type-safe code manipulation approaches focusing on multi-stage programming usually do not permit the inspection of existing code (the *purely generative* approach), or they lose well-typed and well-scoped guarantees while doing so, like in LMS [Rompf and Odersky, 2010]. While purely generative staging is more powerful than one may think, especially when coupled with effects [Kameyama et al., 2015, Kiselyov et al., 2016], our experience in using these and related systems is that code analysis and transformation using first-class code inspection result in programs which are often simpler and more concise, and therefore also easier to write and understand. This aspect is explained in more detail in Chapter 4.

Guarantees about manipulated programs have been encoded via the host language’s type system using techniques such as generalized algebraic data types (GADTs) [Xi et al., 2003, Cheney and Hinze, 2003], higher-order abstract syntax (HOAS) [Pfenning and Elliott, 1988], applicative functors and monads [Kameyama et al., 2015], and De Bruijn indices [Carette et al., 2009, Sheard et al., 2005, Chen and Xi, 2005]. As we have seen in Section 1.2.1, these are often heavyweight approaches — they impose a significant cost on domain experts, who have to deal with complicated type encodings, whereas they would prefer to just express code transformations as simple rewrite rules. In particular, we found that GADTs are hard to manipulate in systems like Haskell and Scala [Giarrusso, 2013, Rompf, 2016].

“Type-based embedding” systems like LMS [Rompf and Odersky, 2010, Rompf, 2016] use implicit conversions to compose code fragments with minimal visual clutter, but it is a complicated and hard-to-understand approach [Jovanovic et al., 2014] which is also not applicable to code pattern-matching (at least in Scala), and requires the manual definition of lots of implicit conversion machinery (see also Section 5.8.2).

1.7.4 Program Transformation

Stratego [Visser, 2002] is a system of composable program transformations that can express rewrite rules using the concrete object syntax, which makes it closely related to quasiquote-based approaches. The main difference with our approach is that Stratego deals with external DSLs, and that its transformations are not statically typed, so they only offer syntactic guarantees about generated programs. Several approaches base program analysis and transformation on variants of the visitor pattern [Hudak, 1998, Ureche et al., 2015]. They are appropriate for a certain range of transformations that only access one level of program trees, but scale poorly to more advanced use-cases.²⁴ Being able to pattern-match and discover the shape of subprograms, which cannot be easily emulated with the visitor pattern, is an invaluable asset,

²⁴As discussed in private communication with the main author [Ureche et al., 2015], March 2016.

making analyses and rewritings both concise and powerful. GHC rewrite rules [Peyton Jones et al., 2001] provide a simple interface for domain experts to write transformations, but they are limited to simple rewritings (syntax expansion similar to hygienic macros in Scheme).

In this chapter, we have focused on manipulating *expressions*. More general approaches to program transformation also deal with higher-level constructs like classes and modules. We see how Squid can be generalized to manipulating classes through *staged classes* in Chapter 8.

2 Application: A Polymorphic Yet Efficient Linear Algebra Library

Many different data analytics tasks boil down to linear algebra primitives. However, in practice data scientists often use specialised libraries for each different type of workload. This is mainly motivated by performance concerns, as specialized libraries are often much more efficient than generic ones.

In this chapter, we present PILATUS, a polymorphic iterative linear algebra language, applicable to various types of data analytics workloads. PILATUS relies on Squid to achieve good performance despite its polymorphic nature.

The design of the PILATUS domain-specific language is inspired by both mathematics and programming languages: its basic constructs are borrowed from abstract algebra, while the key technology behind its polymorphic design is the *tagless final* approach (a.k.a. polymorphic embedding/object algebras).

This design enables us to change the behaviour of arithmetic operations to express matrix algebra, graph algorithms, logical probabilistic programs, and differentiable programs.

Crucially, the polymorphic design of PILATUS allows us to use multi-stage programming and rewrite-based optimisation to recover the performance of specialised code, supporting fixed sized matrices, algebraic optimisations, and fusion.

2.1 Introduction

It is well-known that many problems can be formulated using linear algebra primitives. These problems come from various data analytics domains including machine learning, statistical data analytics, signal processing, graph processing, computer vision, and robotics.

Despite the fact that all these workloads could use a standard unified linear algebra library, in practice many different specialised libraries are developed and used for each of these workload types [Carette and Kiselyov, 2011]. This is mainly due to the performance-critical nature of such data analytics workloads: in order to satisfy their performance requirement, such

workloads use hand-tuned specialised libraries implemented using either general-purpose or specialised domain-specific programming languages.

In this chapter, we demonstrate the PILATUS language (Polymorphic Iterative Linear Algebra, Typed, Universal, and Staged). PILATUS is a *polymorphic* domain-specific language (DSL), in the sense that it can support various workloads, such as standard iterative linear algebra tasks, graph processing algorithms, logical probabilistic programs, and linear algebra programs relying on automatic differentiation. By default, this polymorphic nature causes a significant performance overhead. We demonstrate how to remove this overhead by using safe high-level meta-programming and compilation techniques, and more specifically multi-stage programming (MSP, or *staging*) [Taha and Sheard, 2000, Taha, 2004].

This chapter uses the tagless final approach [Kiselyov, 2012, Carette et al., 2009] (also known as polymorphic embedding [Hofer et al., 2008] and object algebras [Oliveira and Cook, 2012]) in order to embed [Hudak, 1996] the PILATUS DSL in the Scala programming language. This technique allows embedding an *object* language in a *host* language in a type-safe manner. In addition, this approach allows multiple semantics for the embedded DSL (EDSL). Based on this feature and by carefully choosing the abstractions involved in defining PILATUS (such as semi-ring/ring, module, and linear map structures), we provide several evaluation semantics. More specifically, we allow several variants of a linear algebra language, such as: a standard matrix algebra language, a graph language for expressing all-pairs reachability and shortest path problems, a logical probabilistic programming language, and a differentiable programming language. The polymorphic aspect of PILATUS is also essential for the seamless application of staging, and to express different optimised staged variants: fixed size matrices, deforestation [Wadler, 1988, Gill et al., 1993, Svenningsson, 2002, Coutts et al., 2007], and algebraic optimisations.¹

Next, we motivate the need for PILATUS (Section 2.2), and we make the following contributions:

- We present PILATUS, a polymorphic EDSL in Section 2.3. This DSL uses the notion of semi-rings and rings (Section 2.3.2) in order to define operations on each individual element of a vector and a matrix. Furthermore, PILATUS uses the notion of pull arrays (Section 2.3.5) for defining a collection (or array) of elements.
- We present four different languages that are implemented by providing a concrete interpreter for PILATUS in Section 2.5: (1) a standard matrix algebra language (Section 2.5.1); (2) a graph DSL (Section 2.5.2); (3) a logical probabilistic linear algebra language (Section 2.5.3); and (4) a differentiable linear algebra language (Section 2.5.4).

¹We used Scala as the implementation language for PILATUS, but other programming languages with support for lambda expressions and multi-stage programming could be used as well; most of the techniques presented in this chapter can also be implemented in Haskell, OCaml, and Java for example. For expressing rewrite-based optimisations, either the multi-stage programming framework should support code inspection (as is the case with Squid [Parreaux et al., 2017b], which we use), or the developer is responsible for implementing/extending the intermediate representations (as with frameworks like LMS [Rompf and Odersky, 2010]).

- We present our use of multi-stage programming to improve the performance of PILATUS programs by creating a staged language (Section 2.6.2) for fixed size matrices (Section 2.6.5), performing algebraic optimisations (Section 2.6.4), and performing fusion (Section 2.6.6).
- We show the impact of using multi-stage programming on the performance of applications written using PILATUS in Section 2.7. Overall, the implementation of PILATUS consists of around 400 LoC supporting all the features presented in this chapter. PILATUS uses the Squid [Parreaux et al., 2017b] type-safe meta-programming framework for its multi-stage programming facilities, which is the only external library dependency.

Finally, we present the related work in Section 2.8 and conclude the chapter in Section 2.9.

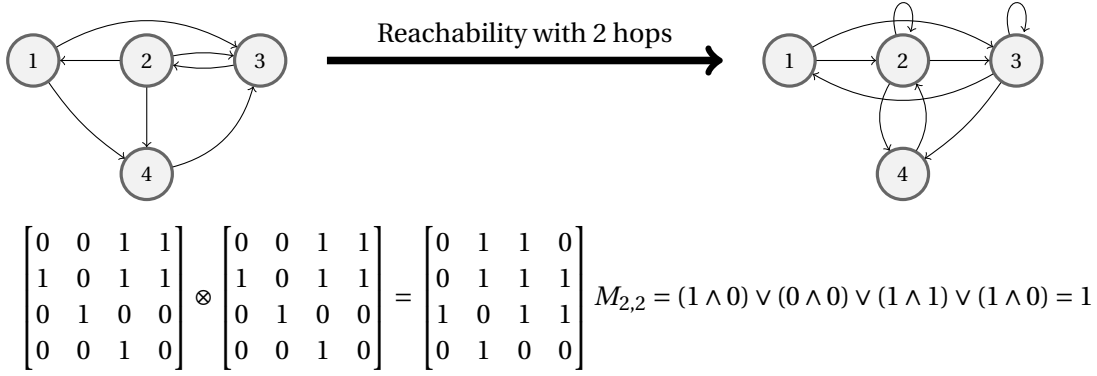
2.2 Motivation

Apart from standard matrix algebra tasks, many numerical workloads in various domains can be expressed using linear algebra primitives [Dolan, 2013]. Among such examples are various graph problems such as reachability and shortest path. Figure 2.1 shows as example the reachability problem on both deterministic and probabilistic graphs.

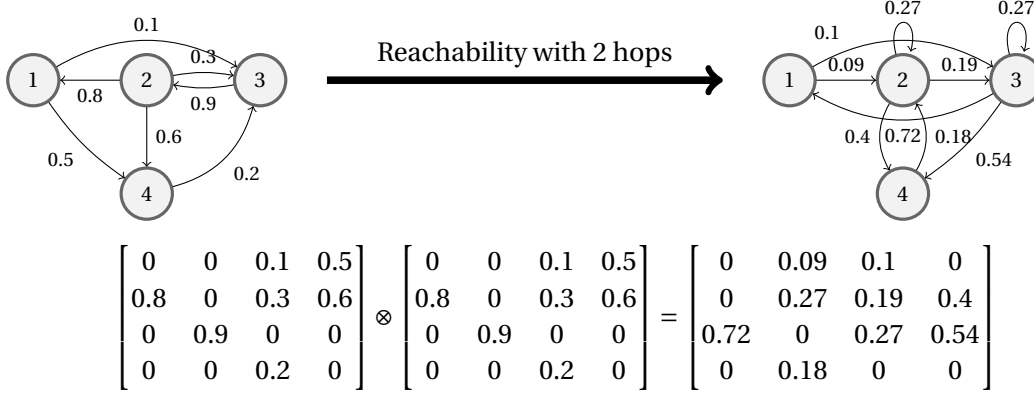
Despite the expressiveness of linear algebra, there are many different libraries specialized for each particular data analytics task. This is because of two main reasons. First, most existing linear algebra libraries do not define the interfaces for extending their usage for the problems in other domains. Second, despite some efforts on providing abstract and extensible linear algebra libraries [Dolan, 2013], such analytical tasks are performance critical. As a result, there should be hand-tuned and specialized libraries for each particular task. As an example, for graph problems, rather than having the linear-algebra-based solutions presented in Figure 2.1, the library developers prefer to provide specialized graph libraries for performance reasons.

This chapter aims to solve both these issues by combining ideas from mathematics and programming languages. The first issue is tackled by defining a polymorphic linear algebra language by using abstractions from abstract algebra, including the ring, module, and linear map structures for expressing scalar values, a vector of values, and a matrix of values, respectively. Furthermore, for implementing these abstract interfaces, we use the tagless-final approach [Carette et al., 2009, Kiselyov, 2012], a well-known technique from the programming language community.

The examples of Figure 2.1 show matrices of elements of various types, for which the addition and multiplication operations can be assigned various meanings. Figure 2.1a shows the usage of linear algebra primitives for expressing graph reachability problems. To do so, the addition and multiplication operators are instantiated to boolean disjunction and conjunction, respectively. For expressing the reachability problems on probabilistic graphs, these two operators are instantiated with the disjunction and conjunction on boolean distributions, as



(a) The reachability problem in a graph can be expressed using matrix-matrix multiplication of the adjacency matrix of a graph. Instead of using the standard addition operator, here we use the boolean disjunction, and instead of the multiplication operator, we use the boolean conjunction.



(b) The reachability problem in a probabilistic graph can also be expressed using matrix-matrix multiplication of its adjacency matrix. Each element of the adjacency matrix represents the presence of a node with probability p . The addition and multiplication operators correspond to disjunction and conjunction of two boolean distributions, respectively.

$$f(x) = \begin{bmatrix} 0 & 0 & 1 & x+3 \\ 8 & 0 & 2x-1 & 6 \\ 0 & 3x+3 & 0 & 0 \\ 0 & 0 & x & 0 \end{bmatrix}^2 = \begin{bmatrix} 0 & 3x+3 & x^2+3x & 0 \\ 0 & 6x^2+3x-3 & 6x+8 & 8x+24 \\ 24x+24 & 0 & 6x^2+3x-3 & 18x+18 \\ 0 & 3x^2+3x & 0 & 0 \end{bmatrix}$$

$$f'(x) = \begin{bmatrix} 0 & 3 & 2x+3 & 0 \\ 0 & 12x+3 & 6 & 8 \\ 24 & 0 & 12x+3 & 18 \\ 0 & 6x+3 & 0 & 0 \end{bmatrix} \quad f'(2) = \begin{bmatrix} 0 & 9 & 10 & 0 \\ 0 & 27 & 20 & 40 \\ 72 & 0 & 27 & 54 \\ 0 & 18 & 0 & 0 \end{bmatrix} \quad f'(2) = \begin{bmatrix} 0 & 3 & 7 & 0 \\ 0 & 27 & 6 & 8 \\ 24 & 0 & 27 & 18 \\ 0 & 15 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 \triangleright 0 & 0 \triangleright 0 & 1 \triangleright 0 & 5 \triangleright 1 \\ 8 \triangleright 0 & 0 \triangleright 0 & 3 \triangleright 2 & 6 \triangleright 0 \\ 0 \triangleright 0 & 9 \triangleright 3 & 0 \triangleright 0 & 0 \triangleright 0 \\ 0 \triangleright 0 & 0 \triangleright 0 & 2 \triangleright 1 & 0 \triangleright 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \triangleright 0 & 0 \triangleright 0 & 1 \triangleright 0 & 5 \triangleright 1 \\ 8 \triangleright 0 & 0 \triangleright 0 & 3 \triangleright 2 & 6 \triangleright 0 \\ 0 \triangleright 0 & 9 \triangleright 3 & 0 \triangleright 0 & 0 \triangleright 0 \\ 0 \triangleright 0 & 0 \triangleright 0 & 2 \triangleright 1 & 0 \triangleright 0 \end{bmatrix} = \begin{bmatrix} 0 \triangleright 0 & 9 \triangleright 3 & 10 \triangleright 7 & 0 \triangleright 0 \\ 0 \triangleright 0 & 27 \triangleright 27 & 20 \triangleright 6 & 40 \triangleright 8 \\ 72 \triangleright 24 & 0 \triangleright 0 & 27 \triangleright 27 & 54 \triangleright 18 \\ 0 \triangleright 0 & 18 \triangleright 15 & 0 \triangleright 0 & 0 \triangleright 0 \end{bmatrix}$$

(c) The derivative of a matrix with respect to the variable x can also be expressed using linear algebra operations. The dual number technique represents each element of a matrix as the pair $v \triangleright d$ of the actual value v and the value of its derivative d . Accordingly, the addition and multiplication operators are the corresponding ones on dual numbers.

Figure 2.1 – Example of problems expressed using different interpretations of linear algebra primitives.

shown in Figure 2.1b. Finally, Figure 2.1c shows the process of computing the derivative of an example matrix expression with respect to a given variable. To do so, each element of the matrix should be represented as a pair of numbers, known as *dual numbers*, where the first component is the actual value of that expression and the second component is the value of its derivative with respect to the given variable. As an example, the dual number representation for the element of the 2^{nd} row and 3^{rd} column is represented as $3 \triangleright 2$, meaning that the actual value of this element at $x = 2$ is $2x - 1 = 3$, whereas its derivative value is $(2x - 1)' = 2$. Similarly, the addition and multiplication operators are instantiated with the corresponding ones operating on dual numbers, which implement the derivative rules.

All of these use cases can be easily represented as PILATUS programs, parameterized over the meaning one wants to use for a particular domain.

The second issue is the performance overhead caused by the polymorphic nature of the language, due to the abstractions introduced in order to solve the first issue. We use multi-stage programming (also known as *staging*) to compile away the overhead corresponding to these abstractions. Moreover, by using a staging framework with support for rewriting, we can also implement algebraic optimization rules for further improving performance.

Next, we give more details on the design of PILATUS.

2.3 PILATUS Design

In this section, we first give an overview of the tagless final approach. Then, we define the polymorphic interface for the semi-ring and ring structures. Afterwards, we show an abstract interface for vectors and matrices using the mathematical notions of modules and linear maps. Finally, we define the interface for a functional encoding of an array of elements and control-flow constructs.

2.3.1 Tagless Final

Tagless final [Carette et al., 2009, Kiselyov, 2012] (also known as polymorphic embedding [Hofer et al., 2008] and object algebras [Oliveira and Cook, 2012] in the context of object-oriented programming languages) is a type-safe approach for *embedding* [Hudak, 1996] domain-specific languages. This approach solves the expression problem [Wadler, 1998] by encoding each DSL construct as a separate function, and leaving their interpretation abstract.

There are different ways of implementing this approach: (1) in languages like Haskell, one can use type classes [Carette et al., 2009, Kiselyov, 2012]; (2) in OCaml, one can use the module system [Carette et al., 2009, Kiselyov, 2018]; (3) in languages like Java, one can use the object-oriented features [Oliveira and Cook, 2012]; and (4) in Scala one can use either type classes or mixin composition (also known as the cake pattern) [Hofer et al., 2008, Rompf and Odersky, 2010].

Chapter 2. Application: A Polymorphic Yet Efficient Linear Algebra Library

In this chapter, we follow the approach based on type classes. Consider a DSL with two constructs, one for creating an integer literal, and the other for adding two terms. The tagless final interface for this DSL is as follows:

```
trait SimpleDSL[Repr] {  
  def lit(i: Int): Repr  
  def add(a: Repr, b: Repr): Repr  
}
```

The code above defines a *trait* (similar to an *interface* in Java or a *module signature* in ML). The SimpleDSL trait is parameterised with a Repr type, which is the type of the objects manipulated by the DSL. This trait contains one abstract method for each constructs of the DSL, here `lit` and `add`.

For convenience, we also typically define free-standing functions for writing programs in the DSL while omitting the particular DSL implementation used:

```
def lit[Repr](i: Int)          (implicit dsl: SimpleDSL[Repr]): Repr =  
  dsl.lit(i)  
def add[Repr](a: Repr, b: Repr)(implicit dsl: SimpleDSL[Repr]): Repr =  
  dsl.add(a, b)
```

These functions require an *implicit* instance of the SimpleDSL trait, and redirect to the implementations of the corresponding methods in that instance. In Scala, implicit parameters need not be specified by users at each call site; indeed, they can be *filled in* automatically by the compiler, based on their expected type. Implicits are the mechanism used to implement type classes in Scala [Oliveira et al., 2010].

One can then define generic programs in the DSL, as follows:

```
def myProgram[Repr](implicit dsl: SimpleDSL[Repr]) = add(lit(2), lit(3))
```

Which can also be written using the following shorthand syntax:

```
def myProgram[Repr: SimpleDSL] = add(lit(2), lit(3))
```

Then, one can specify a particular evaluation semantics for this program. As an example, the following type class instance defines an evaluator/interpreter for SimpleDSL:

```
implicit object SimpleDSLInter extends SimpleDSL[Int] {  
  def lit(i: Int): Int = i  
  def add(a: Int, b: Int): Int = a + b  
}
```


Evaluating the example program above in the REPL with this evaluation semantics, which is automatically picked up by the compiler based on the requested type `Int`, results in:

```
scala> myProgram[Int]
result: Int = 5
```

Rather than directly *evaluating* DSL programs, one can also represent the programs as strings. Below is a type class instance that *stringifies* programs in our DSL:²

```
implicit object SimpleDSLStringify extends SimpleDSL[String] {
  def lit(i: Int): String = i.toString
  def add(a: String, b: String): String = s"$a + $b"
}
```

Evaluating the same program with the stringification evaluation semantics results in:

```
scala> myProgram[String]
result: String = "2 + 3"
```

PILATUS defines a separate type class for each category of the language constructs (e.g., semi-rings, rings, modules, linear maps, etc.), as we show next. We will introduce different evaluation semantics for this DSL by providing type class instances. These evaluation semantics are both interpretation-based (cf. Section 2.5) and compilation-based (cf. Section 2.6).

2.3.2 Semi-Ring and Ring

A semi-ring is defined as a set of numerical values R , with two binary operators $+$ and \times , and two elements 0 (additive identity) and 1 (multiplicative identity), such that for all elements a , b , and c in R the following properties hold:

- $a + 0 = a$
- $a + b = b + a$
- $(a + b) + c = a + (b + c)$
- $a \times 1 = 1 \times a = a$
- $a \times 0 = 0 \times a = 0$
- $(a \times b) \times c = a \times (b \times c)$
- $a \times (b + c) = (a \times b) + (a \times c)$

²String interpolation syntax `s"...$x..."` is equivalent to `"..." + x + "..."`.

```
trait SemiRing[R] {
  def add(a: R, b: R): R
  def mult(a: R, b: R): R
  def one: R
  def zero: R
}

trait Ring[R] extends SemiRing[R] {
  def neg(a: R): R
  def sub(a: R, b: R): R = add(a, neg(b))
}

object Pilatus {
  def add[R](a: R, b: R)(implicit sr: SemiRing[R]): R = sr.add(a, b)
  // ... other boilerplate methods elided for brevity
}
```

Figure 2.2 – The tagless final interface for semi-rings and rings.

- $(a + b) \times c = (a \times c) + (b \times c)$

A ring is a semi-ring with an additional additive inverse operator ($-$) such that for all elements a in R , $a + (-a) = 0$. The binary operator for subtraction can be easily defined as $a - b = a + (-b)$.

The tagless final encoding of semi-rings and rings is shown in Figure 2.2. There are six DSL constructs corresponding to addition, multiplication, negation, subtraction, one, and zero. These methods are redirected to the implementation of the corresponding operations of the `SemiRing` and `Ring` type classes. The implementation of the methods of these type classes are left abstract. These definitions will be given by each concrete semantics, which should make sure that the aforementioned properties hold for the elements of type R .

2.3.3 Module

A mathematical module is a generalization of the notion of a vector space. A module over a particular semi-ring is realised using an addition operator for two modules (similar to vector addition), and a multiplication between a semi-ring element and the module (similar to scalar-vector multiplication). For all elements a and b in a semi-ring R with the multiplicative identity 1_R , and the elements u and v in a (left-)module M , the following properties hold:

- $a \cdot (u + v) = a \cdot u + a \cdot v$
- $(a + b) \cdot u = a \cdot u + b \cdot u$

```

trait Module[V, R, D] {
  implicit val sr: SemiRing[R]
  implicit val dr: SemiRing[D]
  def dim(a: V): D
  def add(a: V, b: V): V
  def smult(s: R, a: V): V
}

object Pilatus {
  // ...
  def dim[V, D](a: V)(implicit m: Module[V, _, D]): D = m.dim(a)
  // ... other boilerplate methods elided for brevity
}

```

Figure 2.3 – The tagless final interface for modules.

- $(a \times b) \cdot u = a \cdot (b \cdot u)$
- $1_R \cdot u = u$

Additionally, the dimension of a finite module generalises the notion of the number of basis vectors³ representing a vector.

Figure 2.3 shows the tagless final interface for modules. The `Module` type class has three type parameters: (1) `V` specifies the type of the underlying vector representation; (2) `R` specifies the type of each element of the vector; and (3) `D` specifies the type of the dimension of the underlying vector. Note that all the elements of type `R` and `D` support semi-ring operations, thanks to the two type class instances `sr` and `dr`. Furthermore, the `Module` type class supports the following operations: (1) the `dim` method returns the dimension of the given module; (2) the `add` method computes the result of the addition of two module elements; and (3) the `smult` method computes the multiplication of a semi-ring element and a given module.

2.3.4 Linear Map

A linear map is a transformation between two modules, which preserves the addition and the scalar multiplication operations of the given module. Assume the linear map M transforming module V to module W , and both modules are over the semi-ring R . Then for all elements f in the linear map M , u and v from module V , and a from the semi-ring R , the following properties hold:

³The basis vectors are *linearly independent* vectors (none of them can be expressed as a linear combination of the other ones) that can be used to express every vector as a *unique linear combination* of them.

```
trait LinearMap[M, V, R, D] {  
  implicit val rowModule: Module[V, R, D]  
  implicit val sr: SemiRing[R]  
  implicit val dr: SemiRing[D]  
  def apply(m: M, v: V): V  
  def compose(m1: M, m2: M): M  
  def add(m1: M, m2: M): M  
  def dims(mat: M): (D, D)  
}  
  
object Pilatus {  
  // ...  
  def apply[M, V](m: M, v: V)(implicit lm: LinearMap[M, V, _, _]): V =  
    lm.apply(m, v)  
  // ... other boilerplate methods elided for brevity  
}
```

Figure 2.4 – The tagless final interface for linear maps.

- $f(u + v) = f(u) + f(v)$
- $f(a \cdot u) = a \cdot f(u)$

Similar to functions, linear maps have two operations. First, a linear map can be applied to a module returning a transformed module, behaving similar to the function application. Second, a linear map can be composed with another linear map resulting in another linear map, behaving similarly to function composition.

Figure 2.4 shows the tagless final encoding of linear maps. Here, we only consider finite linear maps transforming two finite modules, and we assume that both modules are over the same semi-ring (represented with type `R`, and the `sr` type class instance) with the same module type representation (represented with the type `V`). From a vector/matrix point of view, the `compose` and `apply` methods correspond to the matrix-matrix and matrix-vector multiplication, respectively. The `add` method corresponds to the matrix addition operator, and the `dims` construct returns the dimension of the input and output modules, which is represented as a tuple.

2.3.5 Pull Array and Control-Flow Constructs

Using a pull array is a well-known approach in the high-performance functional programming community for a functional encoding of arrays [Svensson and Svenningsson, 2014, Anker and Svenningsson, 2013, Claessen et al., 2012]. In this representation, an array is defined using two

```

trait PullArrayOps[A, E, L] {
  def build(len: L)(f: L => E): A
  def get(arr: A)(i: L): E
  def length(arr: A): L
}
trait Looping[L] {
  def forloop[S](z: S)(n: L)(f: (S, L) => S): S
}
object Pilatus {
  // ...
  def build[A, E, L](len: L)(f: L => E)(implicit p: PullArrayOps[A, E, L]): A =
    p.build(len)(f)
  // ... other boilerplate methods elided for brevity
}

```

Figure 2.5 – The tagless final interface for pull arrays and control-flow constructs.

components: (1) the length of the array; and (2) a function mapping an index to the value of the corresponding element in that array.

Figure 2.5 demonstrates the tagless final encoding of pull arrays and looping constructs. The `build` method is responsible for constructing a pull array of size `len`, in which the i^{th} element is `f(i)`, indexed from 0 to `len - 1`. The `get` method returns the i^{th} element of the array `arr`, whereas the `length` method returns the size of the given array. Finally, the `forloop` method is meant for implementing recursion and iteration. More specifically, this function starts from the state `z`, iterates `n` times (from 0 to `n - 1`), and at the i^{th} step, updates the state `s` with `f(s, i)`.

2.4 Matrix Algebra

In this section, we build the constructs of matrix algebra based on the mathematical notions explained in the previous section. First, we show the construction of vector constructs using modules and pull arrays. Then, we demonstrate the matrix constructs by using linear maps and vectors.

2.4.1 Vector: Module + Pull Array

A vector (more specifically, a dense vector where most elements are non-zero) can be seen as a module the elements of which are stored as a pull array. Given that each element of a vector form a semi-ring, we can define the addition, element-wise multiplication, and dot product of two vectors.

```
trait Vector[V, R, D] extends Module[V, R, D] {
  implicit val pa: PullArrayOps[V, R, D]
  implicit val looping: Looping[D]
  def dim(a: V): D = pa.length(a)
  def add(v1: V, v2: V): V = zipMap(v1, v2, sr.add)
  def smult(s: R, a: V): V = map(a, e => sr.mult(s, e))
  def map(v: V, op: R => R): V = pa.build(pa.length(v))(i => op(pa.get(v)(i)))
  def zipMap(v1: V, v2: V, op: (R, R) => R): V =
    pa.build(pa.length(v1))(i => op(pa.get(v1)(i), pa.get(v2)(i)))
  def elemMult(v1: V, v2: V): V = zipMap(v1, v2, sr.mult)
  def dot(v1: V, v2: V): R =
    looping.forloop(sr.zero)(pa.length(v1))((acc, i) =>
      sr.add(acc, sr.mult(pa.get(v1)(i), pa.get(v2)(i))))
  /* sum and norm are omitted for brevity */
}
```

Figure 2.6 – The tagless final implementation for (dense) vectors.

The implementation for the tagless final encoding of a vector, as well as the mentioned methods are given in Figure 2.6. The `V` type parameter specifies the underlying vector type representation, the `R` type parameter specifies the type of each element of the vector, and the `D` type parameter is the type of the dimension of the underlying vector.

The `zipMap` method, receives two vectors `v1` and `v2` as input and creates a vector of the same size,⁴ for which each element is constructed by applying the binary operator `op` on the corresponding elements from `v1` and `v2`. The `add` and `elemMult` are constructed by passing the addition and multiplication functions of the underlying semi-ring of elements to the `zipMap` method. The `map` method applies a given function to each element of the input vector and produces a vector of the same size with the transformed elements as output. The `smult` method is implemented using this method. Finally, the `dot` method computes the dot product of two vectors `v1` and `v2` by first computing the element-wise multiplication of these vectors, and then adding the elements of this intermediate vector.

Next, we use the mentioned vector data structure together with linear maps in order to define a matrix data-structure.

2.4.2 Matrix: Linear Map + Vector

Figure 2.7 shows the implementation of matrices (more specifically, dense matrices) using linear maps and vectors. The `M` type parameter specifies the type of the underlying matrix

⁴We assume that the input vectors have the same size for the sake of simplicity. In practice, this property can be enforced statically using Scala's powerful implicit programming capabilities, and singleton types.

representation, V represents the type of each row-vector and column-vector of the matrix, R denotes the type of each element of the matrix, and D specifies the type of the dimension of each row and each column of the matrix.

In order to facilitate usages of the generic library, we have implemented several helper methods. The `get` method returns the corresponding element in the r^{th} row and c^{th} column of the matrix.⁵ The `numRows` and `numCols` methods return the number of rows and columns of a matrix, respectively. The `getRow` method returns the vector representing the r^{th} row of the given matrix, whereas `getCol` returns a vector containing the elements in the c^{th} column of the given matrix. Finally, the `zipMap` and `map` methods have similar behaviour to the methods with the same name from the vector data type.

The `add` method returns the result of the addition of two matrices, which is implemented using the `zipMap` method. The `mult` method returns the matrix-matrix multiplication of two matrices. This method is implemented by performing a vector dot-product of each row of the first matrix with each column of the second matrix. Finally, the `transpose` method returns the transpose of the given matrix.

2.4.3 Putting It All Together

Before showing different evaluation semantics in the upcoming sections, we need a way to print the result values. To do so, we define the `Printable` type class which converts the value of a particular type into a string. Figure 2.8 shows the corresponding tagless final definition.

Example. Throughout this chapter, we use the following example matrix program, where we change the values for matrix `m` based on the evaluation semantic that we are interested in:

```
def example[M, D](m: M)(implicit mev: Matrix[M,_,_,D], pev: Printable[M]):
  Unit = {
    import mev._
    val I = eye(numRows(m))
    val m2 = mult(m, m)
    val res = add(I, add(m, m2))
    println(getString(res))
  }
```

This program accepts the matrix `m`, the value of which differs based on the evaluation semantic that we would like to use. The result of this program is the addition of the identity matrix (represented using the `eye` method), the given input matrix, and the second power of it.

In the next sections, we give several concrete interpretations for `PILATUS`, and we show the output of the example program above for each of the interpretations.

⁵As we will see in Section 2.6, `r` and `c` can have types other than `Int`.

```
trait Matrix[M, V, R, D] extends LinearMap[M, V, R, D] {
  implicit val paMat: PullArrayOps[M, V, D]
  implicit val vector: Vector[V, R, D]
  /* Other implicit values: paRow, rowModule, looping, dr, sr */
  /* apply and compose methods use the mult method, elided for brevity. */
  def get(mat: M, r: D, c: D): R =
    paRow.get(paMat.get(mat)(r))(c)
  def numRows(mat: M) = dims(mat)._1
  def numCols(mat: M) = dims(mat)._2
  def getRow(mat: M, i: D): V = paMat.get(mat)(i)
  def getCol(mat: M, j: D): V = getRow(transpose(mat), j)
  def zipMap(m1: M, m2: M, bop: (R, R) => R): M =
    paMat.build(numRows(m1))(i =>
      paRow.build(numCols(m1))(j =>
        bop(get(m1, i, j), get(m2, i, j))))
  def add(m1: M, m2: M): M = zipMap(m1, m2, sr.add)
  def mult(m1: M, m2: M): M =
    paMat.build(numRows(m1))(i =>
      paRow.build(numCols(m2))(j =>
        vector.dot(getRow(m1, i), getCol(m2, j))))
  def transpose(mat: M): M =
    paMat.build(numCols(mat))(i =>
      paRow.build(numRows(mat))(j => get(mat, j, i)))
  /* map, eye, fill, and zeros are omitted for brevity */
}
```

Figure 2.7 – The tagless final implementation for (dense) matrices.

2.5 Interpreted Languages

In this section, we first show an evaluation strategy which results in a standard matrix algebra library. Then, we show how we can define an alternative interpretation which leads to treating PILATUS as a graph library. Afterwards, we show a linear algebra library for logical probabilistic programming. Finally, we demonstrate how PILATUS can behave as a library for differentiable programming.

2.5.1 Standard Matrix Algebra

In order to define a standard matrix algebra library for PILATUS, we start by defining a normal interpreter for rings. Figure 2.9 shows the interpretation for a ring of integer and double values. In both cases, the addition and multiplication operations are defined using the primitive


```

trait Printable[T] {
  def string(e: T): String
}
object Pilatus {
  // ...
  def getString[T](e: T)(implicit p: Printable[T]) = p.string(e)
}

```

Figure 2.8 – The tagless interface for the stringification of the values of different evaluation semantics.

```

implicit object RingInt extends Ring[Int] {
  def add(a: Int, b: Int) = a + b
  def mult(a: Int, b: Int) = a * b
  def one: Int = 1
  def zero: Int = 0
  def neg(a: Int): Int = -a
}
implicit object RingDouble extends Ring[Double] {
  /* Similar to RingInt */
}

```

Figure 2.9 – The tagless final interpreter (a.k.a. type class instances) for a ring of integer and double values.

operations provided by the Scala language.

Figure 2.10 shows an interpreter for pull arrays, where every constructed pull array is materialised into an array of elements. Hence, retrieving an element and returning the size of the pull array is achieved by returning the corresponding element in the materialised array and the length of the array, respectively. Finally, the implementation of `forloop` is achieved by performing a `foldLeft` on the range of elements from 0 to $n-1$, and passing the initial state and the accumulator function.

An alternative way of interpretation for pull arrays, which avoids the materialisation of the intermediate arrays into a sequence, keeps a data structure which holds the length and the constructor function of each element. This representation is given in Figure 2.11.

Example. The standard matrix algebra interpreter evaluates the example program as follows:

```

import Semantics.{ pullArrayInterOps, loopingInt, ringInt }
val adj = Array(Array(0, 0, 1, 5),

```

```
class PullArrayArrayOps[E: ClassTag] extends PullArrayOps[Array[E], E, Int] {  
  def build(len: Int)(f: Int => E): Array[E] =  
    Array.tabulate(len)(f)  
  def get(arr: Array[E])(i: Int): E =  
    arr(i)  
  def length(arr: Array[E]): Int =  
    arr.length  
}  
class LoopingInt extends Looping[Int] {  
  def forloop[S](f: (S, Int) => S)(z: S)(n: Int): S =  
    (0 until n).foldLeft(z)(f)  
}  
object Semantics {  
  implicit def pullArrayArrayOps[E: ClassTag] = new PullArrayArrayOps[E]  
  implicit val loopingInt = new Looping[Int]  
}
```

Figure 2.10 – The tagless final interpreter for a pull array, represented as a list of elements, and the control-flow constructs.

```
      Array(8, 0, 3, 6),  
      Array(0, 9, 0, 0),  
      Array(0, 0, 2, 0))  
val m = build(4)(i => build(4)(j => adj(i)(j)))  
example(m)  
// output:  
[ [ 1, 9, 11, 5 ]  
 , [ 8, 28, 23, 46 ]  
 , [ 72, 9, 28, 54 ]  
 , [ 0, 18, 2, 1 ] ]
```

Note that the `build` method is redirected to the `build` method of the `PullArrayOps` type class (cf. Figure 2.5).

2.5.2 Graph DSL for Reachability and Shortest Path

A directed graph can be represented using its adjacency matrix. More specifically, a graph with n vertices can be represented using a matrix of size $n \times n$, in which all elements are Boolean. If the element in the i^{th} row and j^{th} column is true, this means that there is an edge between the i^{th} and j^{th} vertices in the graph.

```

case class PullArrayInter[E](len: Int, f: Int => E)

class PullArrayInterOps[E] extends PullArrayOps[PullArrayInter[E], E, Int] {
  def build(len: Int)(f: Int => E): PullArrayInter[E] =
    PullArrayInter(len, f)
  def get(arr: PullArrayInter[E])(i: Int): E =
    arr.f(i)
  def length(arr: PullArrayInter[E]): Int =
    arr.len
}
object Semantics {
  // ...
  implicit def pullArrayInterOps[E] = new PullArrayInterOps[E]
}

```

Figure 2.11 – The tagless final interpreter for a pull array, represented as a pair of length and the element constructor function.

In order to support such adjacency matrices, we need to use the Boolean semi-ring for the matrix elements. Figure 2.12 shows the implementation of the Boolean semi-ring, in which addition performs disjunction, and multiplication performs conjunction.

Using the Boolean semi-ring for the elements of a matrix leads to a graph library. This instantiation of PILATUS is appropriate for expressing reachability computations among all vertices of a graph: given an adjacency matrix M , each element of $M \times M$ shows the existence of a path of length 2 between the two vertices in the corresponding graph.

The graph algorithms that can be implemented on top of PILATUS are not limited to reachability ones. By adding other types of semi-rings, one can express other graph computation problems. As an example, Tropical semi-rings can express shortest-path graph problems [Mohri, 2002, Dolan, 2013]. Figure 2.13 shows the tagless final encoding of Tropical semi-rings. The ShortestPath data type represents the path between two nodes of a graph, where Unreachable specifies a path of length $+\infty$, and Distance(v) specifies a path of length v . The Tropical semi-ring computes the minimum length of two paths as the addition operator of the semi-ring, and adds the length of two paths as the multiplication operator. We omit the definition for other semi-rings for graph and other similar problems (e.g., linear equations, data-flow analysis, petri nets, etc., which are already explored in the literature [Dolan, 2013]).

Example. When one uses the Boolean semi-ring, the example program is actually computing the existence of paths with maximum length two among all the nodes. When we provide the adjacency matrix of the graph of Figure 2.1a, the example program evaluates to:

```
class SemiRingBoolean extends SemiRing[Boolean] {
  def add(a: Boolean, b: Boolean) = a || b
  def mult(a: Boolean, b: Boolean) = a && b
  def one: Boolean = true
  def zero: Boolean = false
}
object Semantics {
  // ...
  implicit val semiRingBoolean = new SemiRingBoolean
}
```

Figure 2.12 – The tagless final interpreter for a semi-ring of Boolean values, used for expressing graph reachability problems.

```
import Semantics.{ pullArrayInterOps, loopingInt, ringInt, semiRingBoolean }
val adj = Array(Array(false, false, true, true),
                 Array(true, false, true, true),
                 Array(false, true, false, false),
                 Array(false, false, true, false))
val m = build(4)(i => build(4)(j => adj(i)(j)))
example(m)
// output:
[ [ T, T, T, T ]
, [ T, T, T, T ]
, [ T, T, T, T ]
, [ F, T, T, T ] ]
```

2.5.3 Probabilistic Linear Algebra Language

Probabilistic models are used in many applications including artificial intelligence, machine learning, cryptography, and economics. Probabilistic programming languages have proven to be successful for expressing such stochastic models in a declarative style without worrying about computational aspects [Carpenter et al., 2017, Gordon et al., 2014, Goodman et al., 2012, Kiselyov and Shan, 2009]. As an example, an important computer vision application was recently expressed in only 50 lines of code in the Picture probabilistic programming language [Kulkarni et al., 2015].

In this chapter, our aim is not to make PILATUS a full-fledged probabilistic programming language. Instead, we show how we can encode Boolean probability distributions in PILATUS in the form of a semi-ring. This means that we support the conjunction and disjunction between two Boolean distributions. Also, the zero and one elements of the semi-ring correspond to the

```

sealed trait ShortestPath {
  def add(o: ShortestPath): ShortestPath = (this, o) match {
    case (Unreachable, x) => Unreachable
    case (x, Unreachable) => Unreachable
    case (Distance(v1), Distance(v2)) => Distance(v1 + v2)
  }
  def min(o: ShortestPath): ShortestPath = (this, o) match {
    case (Unreachable, x) => x
    case (x, Unreachable) => x
    case (Distance(v1), Distance(v2)) => Distance(math.min(v1, v2))
  }
}

case class Distance(v: Int) extends ShortestPath
case object Unreachable extends ShortestPath

class SemiRingTropical extends SemiRing[ShortestPath] {
  def add(a: ShortestPath, b: ShortestPath) = a.min(b)
  def mult(a: ShortestPath, b: ShortestPath) = a.add(b)
  def one: ShortestPath = Distance(0)
  def zero: ShortestPath = Unreachable
}

object Semantics {
  // ...
  implicit val semiRingTropical = new SemiRingTropical
}

```

Figure 2.13 – The tagless final interpreter for a semi-ring of Boolean values, used for expressing graph shortest-path problems.

distribution with the probability of one for false and true, respectively. As a side effect of the compositional design of PILATUS, we can support vectors and matrices of such distributions as well, virtually for free. Thus, PILATUS supports probabilistic graphs and the associated path queries, similar to systems such as ProbLog [De Raedt et al., 2007].

Figure 2.14 shows the tagless final implementation for Boolean distributions. The BoolProb data type has a list of probabilities assigned to each Boolean value. This data type is actually a *probability monad* [Giry, 1982, Erwig and Kollmansberger, 2006]. As is customary with monad implementation in Scala, the flatMap method represents the bind operator of the monad, and the apply method of the companion object represents the unit operator. The normalise method makes sure that the list of probabilities associated to each Boolean value has distinct Boolean values, and that the probabilities sum up to one.

```

case class BoolProb(l: List[(Boolean, Double)]) {
  def flatMap(f: Boolean => BoolProb): BoolProb = {
    val ll = for(x <- l; y <- f(x._1).l) yield { y._1 -> (y._2 * x._2) }
    BoolProb(ll).normalise()
  }
  def normalise(): BoolProb = {
    val sum = l.map(_._2).sum
    val nl = l.groupBy(_._1).mapValues(_._2.map(_._2).sum / sum)
    BoolProb(nl.toList)
  }
}

object BoolProb {
  def apply(v: Boolean): BoolProb = BoolProb(List(v -> 1.0))
}

class SemiRingBoolProb extends SemiRing[BoolProb] {
  def add(a: BoolProb, b: BoolProb) = a.flatMap(x => if(x) one else b)
  def mult(a: BoolProb, b: BoolProb) = a.flatMap(x => if(x) b else zero)
  def one: BoolProb = BoolProb(true)
  def zero: BoolProb = BoolProb(false)
}

object Semantics {
  // ...
  implicit val semiRingBoolProb = new SemiRingBoolProb
}

```

Figure 2.14 – The tagless final implementation using the Boolean probability monad for semi-ring operations.

There are many alternative implementations for the probability monad such as lazy trees [Kiselyov and Shan, 2009] with the possibility to support distributions for values other than Booleans. Furthermore, in this context one can use various optimisations such as variable elimination [Dechter, 1998]. Finally, it is possible to explore other inference mechanisms [Mansinghka et al., 2018]. All these aspects are orthogonal to the purposes of this work, and PILATUS can be extended to support all these features, which we leave as exercises to the reader.

Example. When we give the adjacency matrix of the probabilistic graph of Figure 2.1b as the input to the example program, the evaluation is as follows:

```

import Semantics.{ pullArrayInterOps, loopingInt, ringInt, semiRingBoolProb }
def flip(p: Double): BoolProb = BoolProb(List(true -> p, false -> (1 - p)))

```

```

val adj = Array(Array(flip(0), flip(0), flip(0.1), flip(0.5)),
                 Array(flip(0.8), flip(0), flip(0.3), flip(0.6)),
                 Array(flip(0), flip(0.9), flip(0), flip(0)),
                 Array(flip(0), flip(0), flip(0.2), flip(0)))
val m = build(4)(i => build(4)(j => adj(i)(j)))
example(m)
// output:
[ [ 1, 0.1, 0.2, 0.5 ]
  , [ 0.8, 1, 0.4, 0.8 ]
  , [ 0.7, 0.9, 1, 0.5 ]
  , [ 0, 0.2, 0.2, 1 ] ]

```

As in the previous section, the example program computes the all-pairs path with maximum length of two. Hence, the result matrix is the probability of the existence of a path by traversing at most one intermediate node.

2.5.4 Differentiable Linear Algebra DSL

Many applications in machine learning such as training artificial neural networks require computing the derivative of an objective function. In many cases, the manual derivation of analytical derivatives is not a practical solution, as it is error prone and time consuming. Hence, several techniques were developed for automating the derivation process.

Automatic differentiation (or algorithmic differentiation) is one of the most well-known techniques to systematically compute the derivative of a program. This technique systematically applies the chain rule, and evaluates the derivatives for the primitive arithmetic operations (such as addition, multiplication, etc.) [Baydin et al., 2015a].

Among different implementations of automatic differentiation, here we show the forward mode technique using *dual numbers*. In this implementation, every number is augmented with an additional component, which maintains the computed derivative value. Correspondingly, all primitive operations should be augmented with the appropriate derivation computation.

Figure 2.15 demonstrates the generic tagless final interface for the dual number representation of a ring. This interface uses the pair representation for dual numbers, in which the first component is the normal value, whereas the second component is the derivative value. The second component in the implementation of the addition operator reflects the addition rule of derivation ($d(a + b) = da + db$), whereas the one in multiplication reflects the multiplication rule ($d(a \times b) = da \times b + a \times db$).

Example. Let us consider again the example matrix given in Figure 2.1c. By representing this input matrix using dual numbers, our running example is evaluated as follows:

```
class DualSemiRing[R](implicit val sr: SemiRing[R])
extends SemiRing[(R, R)] {
  type Dual = (R, R)
  def add(a: Dual, b: Dual) = (sr.add(a._1, b._1), sr.add(a._2, b._2))
  def mult(a: Dual, b: Dual) =
    (sr.mult(a._1, b._1), sr.add(sr.mult(a._1, b._2), sr.mult(a._2, b._1)))
  def one: Dual = (sr.one, sr.zero)
  def zero: Dual = (sr.zero, sr.zero)
}

class DualRing[R](implicit val ring: Ring[R])
extends DualSemiRing[R] with Ring[(R, R)] {
  def neg(a: Dual) = (ring.neg(a._1), ring.neg(a._2))
}

object Semantics {
  // ...
  implicit def dualSemiRing[R: SemiRing] = new DualSemiRing[R]
  implicit def dualRing[R: Ring] = new DualRing[R]
}
```

Figure 2.15 – The tagless final implementation using dual numbers for ring operations.

```
import Semantics.{ pullArrayInterOps, loopingInt, ringInt, dualRing }
val adj = Array(Array(0 -> 0, 0 -> 0, 1 -> 0, 5 -> 1),
                 Array(8 -> 0, 0 -> 0, 3 -> 2, 6 -> 0),
                 Array(0 -> 0, 9 -> 3, 0 -> 0, 0 -> 0),
                 Array(0 -> 0, 0 -> 0, 2 -> 1, 0 -> 0))
val m = build(4)(i => build(4)(j => adj(i)(j)))
example(m)
// output:
[ [ 1 -> 0, 9 -> 3, 11 -> 7, 5 -> 1 ]
, [ 8 -> 0, 28 -> 27, 23 -> 8, 46 -> 8 ]
, [ 72 -> 24, 9 -> 3, 28 -> 27, 54 -> 18 ]
, [ 0 -> 0, 18 -> 15, 2 -> 1, 1 -> 0 ] ]
```

More specifically, computing the square of this matrix results in:

```
val m2 = compose(m, m)
println(getString(m2))
// output:
[ [ 0 -> 0, 9 -> 3, 10 -> 7, 0 -> 0 ]
, [ 0 -> 0, 27 -> 27, 20 -> 6, 40 -> 8 ]
, [ 72 -> 24, 0 -> 0, 27 -> 27, 54 -> 18 ]
```



```
, [ 0 -> 0, 18 -> 15, 0 -> 0, 0 -> 0 ] ]
```

This output is the same as what we have observed in Figure 2.1c.

2.6 Staging and Optimisation

In this section, we show how to use multi-stage programming (MSP, or just *staging*) to improve the performance of PILATUS programs, by removing the abstraction overhead incurred by the high-level programming features we use to make our DSL polymorphic.

2.6.1 Augmented Multi-Stage Programming

We start by quickly recapitulating the discipline of *multi-stage programming* in the context of Squid, and its extension to pattern matching and rewriting.

Runtime Compilation. Recall that, as explained in Section 1.3.7, Squid allows programmers to runtime-compile code on the fly, using the `.compile` method, which produces bytecode that can then be run as efficiently as if it had been compiled normally.

Multi-Stage Programming (MSP). The goal of MSP is to turn a program which contains abstractions and indirections into a *code generator* — instead of producing its result directly, the “staged” program will produce *code* that is more straightforward (free of abstractions), to compute the program’s result more efficiently. To achieve this, one annotates the non-static parts of the program (those that should be executed later) using Squid quasiquotes and Code types. Thanks to runtime compilation, the staged program effectively partially evaluates the fixed parts of a program even if they depend on values obtained at runtime.

Code Pattern Matching and Rewriting. Squid extends the capabilities of classical MSP languages by supporting code pattern matching and rewriting (quasiquotes are allowed in patterns, as explained in Chapter 1), which lets programmers inspect already-composed code fragments in a type-safe way. As we will see in Section 2.6.4, in practice this saves programmers the trouble of having to define their own inspectable program representations delaying the production of actual code values.

2.6.2 Staging PILATUS

Thanks to the polymorphic nature of PILATUS, it is quite straightforward to turn a given semantics into a multi-stage program. All we need to do is to provide an evaluation semantics which manipulates program fragments instead of normal values, and which composes these fragments together instead of directly evaluating the results of each operation.

Figure 2.16 shows the staged versions of some of the PILATUS interfaces. A `RingCode[T]` is a ring implementation⁶ that manipulates `Code[T]` ring elements (the `RingCode[T]` class extends `Ring[Code[T]]`). Remember from Section 1.3.3 that the `CodeType[T]` type class is used to automatically infer runtime type representations, which is necessary for Squid program manipulation. Notice that the implicit `ringCode` definition takes an implicit argument of type `Code[Ring[T]]`. This works out of the box, because Squid can turn an implicit `Ring[T]` into a `Code[Ring[T]]` automatically, lifting the code used for generating the implicit.

As an example, consider the following polymorphic PILATUS program:

```
def polymorphicProgram[R: Ring](a: R, b: R): R = mult(add(a, one), b)
```

And the following two usages, one with a direct `R = Int` interpretation, and one with a staged `R = Code[Int]` one:

```
import Semantics.{ ringInt }, StagedSemantics.{ ringCode }
Console.print("Enter an integer number: ")
val k = Console.readInt
val f_slow = (x: Int) => polymorphicProgram(x, k)
val f_code = (x: Code[Int]) => polymorphicProgram(x, Const(k))
val f_fast = code"${f_code}".compile
```

The `Const` constructor turns a primitive value (here an `Int`) into a code value (here a `Code[Int]`). Notice that we insert `f_code` into a quasiquote even though it is not a code value, but a function from code to code; in fact, it is implicitly lifted by Squid to the corresponding code value (see Section 1.3.9).

Assuming the user enters the number 27 on the console, the code generated at runtime for `f_fast` will be equivalent to `(x: Int) => Semantics.ringInt.mult(Semantics.ringInt.add(x, 1), 27)` which, after inlining of the statically-dispatched `ringInt` methods, corresponds to `(x: Int) => (x + 1) * 27`. To understand why this is much more efficient than the `f_slow` version, consider that the evaluation of `f_slow` has to go through virtual dispatch of all the ring operations; moreover, it also has to use boxed representations of the manipulated integer values due to the generic context in which ring operations are defined, which requires repeated allocations and unwrapping of boxed integers. As a result, in a realistic workload, even the just-in-time compiler will typically not manage to make that code as fast as the straightforward primitive operations performed by `f_fast`.⁷

⁶Strictly speaking, this implementation does not form a ring, because for example `code"2+1"` is not the same as `code"1+2"` — though they are “morally” equivalent as they represent equivalent programs.

⁷Runtime systems like the CLR for C# avoid boxing by performing runtime specialisation of generic code, but that only achieves a small part of all the optimisation and partial evaluation we are interested in here. C++ templates can perform advanced compile-time specialisation, which could get us closer to our goal (though this means specialisation could not rely on runtime values), but they are difficult and heavyweight, yet much less flexible because they do not allow for first-class manipulation of code values.

This kind of overhead easily compounds as we introduce more abstractions, to the point where non-staged abstract programs end up being *orders of magnitude* slower than the staged versions [Yallop, 2017], as we will see in Section 2.7.

2.6.3 Staged Representation Optimisations

An interesting aspect of MSP is that it lets us define data structures made of partially-staged data. For example, if we want to partially evaluate the allocation of pairs and the selection of their components, we can use representations of type `(Code[A], Code[B])` instead of `Code[(A, B)]`.

This comes in useful when representing dual numbers in our staged interpreter. We can implement an alternative to `DualRing` that is specialised for handling code values, and define its operations accordingly, for example:

```
def mult(a: (Code[R], Code[R]), b: (Code[R], Code[R])) =
  (code"$sr.mult(${a._1}, ${b._1})",
   code"$sr.add($sr.mult(${a._1}, ${b._2}), $sr.mult(${a._2}, ${b._1}))")
```

Note that in the code above, we use program fragments `a._1` and `b._1` *several times*. This is fine, because the default intermediate representation that Squid uses to encode program fragment is based on the A-normal form [Flanagan et al., 1993], which let-binds every subexpression to a local variable, and thus avoids code duplication [Parreaux et al., 2017b,a]; in other words, by inserting a given code value in several places, we only duplicate variable references.

2.6.4 Algebraic Optimisations

Thanks to the staged interpretations of PILATUS, which allows us to manipulate program fragments as first-class values, we can leverage the algebraic properties of ring structures to perform optimisations. To do so, we can *extend* the staged ring implementation, so that we use the normal staged method implementations by default, and *override* those methods where there is a potential for algebraic optimisations. The goal of the overridden methods is to return simplified program fragments based on the shape of their inputs.

This technique is similar to the original tagless final [Carette et al., 2009] and polymorphic embedding [Hofer et al., 2008] approaches to algebraic optimisation. The main difference is that thanks to Squid's analytic capabilities, we do not need to create our own intermediate symbolic representation of programs, and instead we can pattern-match on code values directly.

An implementation of this optimised staged semantics for rings is given in Figure 2.17. When used in pattern position, traditional quasiquote escapes `${...}`, which *insert* code values into bigger expressions, are written `$$${...}` instead.

```
import squid.IR.Predef._ // import the `Code`, `CodeType` and `code`
                           functionalities

class SemiRingCode[T: CodeType](val sr: Code[SemiRing[T]]) extends
  SemiRing[Code[T]] {
  def add(a: Code[T], b: Code[T]) = code"$sr.add($a, $b)"
  def mult(a: Code[T], b: Code[T]) = code"$sr.mult($a, $b)"
  def one: Code[T] = code"$sr.one"
  def zero: Code[T] = code"$sr.zero"
}

class RingCode[T: CodeType](val ring: Code[Ring[T]])
  extends SemiRingCode[T](ring) with Ring[Code[T]] {
  def neg(a: Code[T]) = code"$ring.neg($a)"
}

class PullArrayCodeOps[E: CodeType]
  extends PullArrayOps[Code[PullArrayInter[E]], Code[E], Code[Int]] {
  def build(len: Code[Int])(f: Code[Int] => Code[E]): Code[PullArrayInter[E]] =
    code"$PullArrayInter($len, $f)"
  def get(arr: Code[PullArrayInter[E]])(i: Code[Int]): Code[E] = code"$arr.f($i)"
  def length(arr: Code[PullArrayInter[E]]): Code[Int] = code"$arr.len"
}

object StagedSemantics {
  implicit def semiRingCode[T: CodeType]
    (implicit cde: Code[SemiRing[T]]): SemiRing[Code[T]] = new SemiRingCode(cde)
  implicit def ringCode[T: CodeType]
    (implicit cde: Code[Ring[T]]): Ring[Code[T]] = new RingCode(cde)
  // other similar definitions elided...
}
```

Figure 2.16 – The tagless final encoding of compiled rings, pull arrays, and control-flow constructs.

```

class SemiRingOptCode[T: CodeType](sr: Code[SemiRing[T]]) extends
  SemiRingCode[T](sr) {
  override def add(a: Code[T], b: Code[T]) = (a, b) match {
    case (_, code"$$sr.zero") => a
    case (code"$$sr.zero", _) => b
    case _ => super.add(a, b)
  }
  override def mult(a: Code[T], b: Code[T]) = (a, b) match {
    case (_, code"$$sr.zero") => code"$$sr.zero"
    case (code"$$sr.zero", _) => code"$$sr.zero"
    case (_, code"$$sr.one") => a
    case (code"$$sr.one", _) => b
    case _ => super.mult(a, b)
  }
}

class RingOptCode[T: CodeType](ring: Code[Ring[T]]) extends RingCode[T](ring) {
  override def neg(a: Code[T]) = a match {
    case code"$$ring.zero" => code"$$ring.zero"
    case _ => super.neg(a)
  }
}

```

Figure 2.17 – The tagless final encoding of the compiled library of PILATUS, which applies algebraic optimisations for the elements of matrices.

Many more algebraic rewritings can be added to perform partial evaluation and normalization of program fragments. We have omitted them for the sake of brevity. Furthermore, one can encode the algebraic properties of modules (cf. Section 2.3.3) and linear maps (cf. Section 2.3.4) as rewrite rules, which we leave for the future.

2.6.5 Fixed-Size Matrix DSL

In some applications, such as computer vision, the matrices or vectors have small sizes, and sometimes these sizes are statically known (e.g. a vector of size 3 to show a point in the 3D space). In these cases, the necessary memory for the corresponding arrays can be allocated at compile time (or even stack-allocated), leading to better performance and memory consumption at run time.

PILATUS can be instantiated with an evaluator that makes sure that the length of arrays is known during the compilation time. In this case, the representation of a pull array is a sequence

```
case class PullArrayCode[E](len: Code[Int], f: Code[Int] => Code[E])

class PullArrayCodeFusedOps[E]
  extends PullArrayOps[PullArrayCode[E], Code[E], Code[Int]] {
  def build(len: Code[Int])(f: Code[Int] => Code[E]): PullArrayCode[E] =
    PullArrayCode(len, f)
  def get(arr: PullArrayCode[E])(i: Code[Int]): Code[E] =
    arr.f(i)
  def length(arr: PullArrayCode[E]): Code[Int] =
    arr.len
}
```

Figure 2.18 – The tagless final encoding of the compiled library of PILATUS, which removes all unnecessary intermediate arrays.

of the symbolic representation for each element. Furthermore, the representation for its length is an integer, instead of a symbolic representation. Interestingly, this representation is the same as the one shown in Figure 2.10, but with the E type instantiated to multi-stage code types.

2.6.6 Fused DSL

Deforestation [Wadler, 1988, Gill et al., 1993, Svenningsson, 2002, Coutts et al., 2007] is a well-known technique used in functional languages in order to remove the unnecessary intermediate data structures. This removal has a positive effect on both memory consumption and run-time performance, thanks to the removal of unnecessary memory allocations and avoidance of unnecessary computations.

One of the key advantages of using pull arrays is providing deforestation. However, to benefit from this feature, one should provide an appropriate representation for pull arrays which avoids materialisation. This can be achieved by symbolically maintaining the length and the constructor function. Whenever the array is indexed or the length of array is needed, instead of creating a symbolic representation for them, we can use the maintained length and constructor function.

Figure 2.18 represents the implementation of fused pull array, and a compiler allowing deforestation for PILATUS.

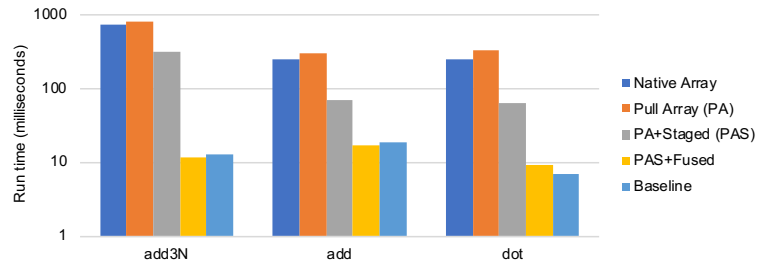


Figure 2.19 – Performance comparison between PILATUS with different configurations and a baseline low-level implementation.

2.7 Evaluation

In this section, we show how multi-stage programming and rewriting can make PILATUS faster than the high-level implementation, while being competitive with a handwritten low-level implementation. We use several micro benchmarks consisting of a pipeline of vector operations such as addition, dot product, and norm. Each benchmark is tested with five different approaches:

- PILATUS by using a native array without optimisation
- PILATUS by using a pull array without optimisation
- PILATUS by using a pull array with staging
- PILATUS by using a pull array with staging and fusion
- A handwritten low-level optimised implementation

The experiments are performed on a six-core Intel Xeon E5-2620 v2 processor with 256GB of DDR3 RAM (1600Mhz), with Scala version 2.12.8 running on the OpenJDK 64-Bit Server VM (build 24.95-b01) with Java 1.7.0_101.

Figure 2.19 shows the performance results. For these experiments, the input vectors are all stored in a native JVM array, consisting of one million integer elements. Based on these experiments we make the following observations. First, changing the usage of native array representation to a pull array causes a minor performance overhead. This is because the JIT of JVM is unable to remove the overhead caused by the lambdas used in a pull array. Second, the overhead of lambdas as well as several other overheads are removed by using staging. This performance improvement is between 2.5x to 5x. Finally, the intermediate arrays are successfully removed by benefiting from the fusion of pull arrays (cf. Section 2.6.6). The improvement varies between 4x to 26x depending on the number of removed intermediate arrays. This makes the staged and fused PILATUS competitive with the baseline low-level implementation.

2.8 Related Work

2.8.1 Linear Algebra Languages and Libraries

The R programming language [R Core Team, 2014] is widely used by statisticians and data miners. It provides a standard language for statistical computing that includes arithmetic, array manipulation, object oriented programming and system calls. It is a Turing-complete language. By contrast, with our language we chose to focus solely on linear algebra operations. This minimalistic approach results in a language that is not Turing Complete, but is nevertheless polymorphic in various dimensions.

The Spiral [Püschel et al., 2005] project introduces the languages SPL [Xiong et al., 2001], OL [Franchetti et al., 2009], and more recently LL [Spampinato and Püschel, 2014] which mainly captures the non-iterative matrix operations of PILATUS. Furthermore, the intermediate languages Σ -SPL [Franchetti et al.] and Σ -LL [Spampinato and Püschel, 2014] expose opportunities to perform loop fusion. One interesting direction is to use the search-based techniques to perform global optimisations offered by Spiral. Furthermore, Spiral in Scala [Ofenbeck et al., 2013] supports loop unrolling and fixed size matrices by using the staging facilities offered by LMS [Rompf and Odersky, 2010] and abstracting over data layout. Kiselyov [2018] has used the tagless final approach and staging facilities of MetaOCaml in order to implement a linear algebra DSL based on rings and pull arrays (but not modules and linear maps) and has implemented many of the optimisations that we have seen in this chapter. However, to the best of our knowledge, none of these projects consider graph algorithms, probabilistic programming, and automatic differentiation of linear algebra.

In the Haskell programming language, Dolan [2013] implements a linear algebra library which uses different semi-ring configurations for expressing graph algorithms, as well as several other algorithms. We can easily extend PILATUS in order to support the additional semi-ring configurations used in that work. While Elliott [2009] implements a library for forward automatic differentiation in Haskell, Dolan does not consider automatic differentiation. Since both of these libraries are implemented without any multi-stage programming facilities, neither can support the staged libraries provided by PILATUS.

2.8.2 Deforestation and Array Fusion

Deforestation [Wadler, 1988] and the corresponding short cut techniques [Gill et al., 1993, Svenningsson, 2002, Coutts et al., 2007] were introduced for functional languages for removing the unnecessary intermediate collections. Recently, these techniques have been implemented as a library using multi-stage programming [Jonnalagedda and Stucki, 2015a, Kiselyov et al., 2017, Shaikhha et al., 2018a].

On the other hand, in the high-performance functional array programming there are the two well-known array representations, which also achieve deforestation: pull arrays and push

arrays [Anker and Svenningsson, 2013, Claessen et al., 2012]. Each one of these two array representations comes with its own benefits, for which [Svensson and Svenningsson, 2014] combines the benefits of these two complementary representations. Pull arrays have been used for various DSLs [Axelsson et al., 2011, Kiselyov, 2018, Shaikhha et al., 2017] to produce efficient low-level code from the high-level specification of linear algebra programs. However, none of these systems consider other domains presented in this chapter.

2.8.3 Automatic Differentiation and Differentiable Programming

Many techniques for finding optima of a given objective function (such as gradient-descent-based techniques) require the derivative of that function. Automatic differentiation (AD) [Kedem, 1980] is one of the key techniques for automatically computing the derivative of a given program. Thus, these tools are an essential component of many machine learning frameworks. There is a large body of work on AD frameworks for imperative programming languages such as Tapenade [Hascoet and Pascual, 2013] for C and Fortran, ADIFOR [Bischof et al., 1996] for Fortran, and Adept [Hogan, 2014] and ADIC [Narayanan et al., 2010] for C++, ADiMat [Bischof et al., 2002], ADiGator [Weinstein and Rao, 2016], and Mad [Forth, 2006] performs AD for MATLAB programs, whereas AutoGrad [Maclaurin et al., 2015], Theano [Bergstra et al., 2010], Tensorflow [Abadi et al., 2016] performs AD for a subset of Python programs. There are also AD tools developed for functional languages such as DiffSharp [Baydin et al., 2015b] for F#, dF^\sim [Shaikhha et al., 2018b] for a subset of F#, Stalingrad [Pearlmutter and Siskind, 2008] for a dialect of Scheme, as well as the work by Karczmarsczuk [1999] and Elliott [2009] for Haskell. The most similar work to ours is Lantern [Wang et al., 2018], which uses the multi-stage programming features provided by LMS [Rompf and Odersky, 2010] in order to perform AD for numerical programs written in a subset of Scala. A key feature provided by Lantern is reverse-mode AD using *delimited continuations* [Danvy and Filinski, 1990]; this can also be supported by PILATUS, but we leave it for future work. All these frameworks focus on differentiable programming, whereas PILATUS also supports graph computations and probabilistic programming, while providing algebraic and low-level optimizations for improved performance.

2.8.4 Probabilistic Programming

Probabilistic programming languages (PPL) can express stochastic models in a productive manner without worrying about the low-level details [Gordon et al., 2014]. Infer.NET [Minka et al., 2014], Picture [Kulkarni et al., 2015], and probabilistic C are examples of imperative PPLs, whereas, BUGS [Gilks et al., 1994], STAN [Carpenter et al., 2017], and Church [Goodman et al., 2012] are functional PPLs. Figaro [Pfeffer, 2009] is an object-oriented probabilistic programming language which is defined as an EDSL on top of Scala. PRISM [Sato, 2008], BLOG [Milch et al., 2007], and ProbLog [De Raedt et al., 2007] are examples of Logical PPLs. PILATUS is inspired by both logical PPLs and the embedding of functional PPLs [Kiselyov and Shan, 2009]. Although PILATUS has a more limited expressivity power in comparison with

other logical probabilistic programs, it supports various other domains such as differentiable programming for linear algebra workloads.

2.9 Conclusions

In this chapter, we have presented PILATUS, a polymorphic linear algebra language. This language is embedded in Scala and can have interpretations supporting various domains, such as standard matrix algebra, all-pairs reachability and shortest-path computations for graphs, logical probabilistic programming, and differentiable programming. In order to compensate for the performance penalty caused by the code abstractions, PILATUS uses multi-stage programming, removing the associated overhead by construction. Furthermore, leveraging the mathematical nature of PILATUS, we use algebraic rewrite rules to further improve the performance.

3 The Modular Implementation of Squid

The Squid metaprogramming framework is implemented as a simple software library, and requires no changes to the Scala compiler. Squid extends Scala's type system solely by relying on its powerful macro system. We now examine how Squid is implemented, leveraging Scala 2's extensive metaprogramming capabilities.

This chapter is particularly concerned with the modular implementation which constitutes the backbone of Squid, and more specifically: 1. how it abstracts over different intermediate representations and interpretations of program fragments thanks to object algebras; 2. how it helps in deeply embedding domain-specific languages; 3. how it virtualizes many constructs of the Scala language to make the handling of program representations easier; and 4. how it leverages the powerful metaprogramming tools already offered by the Scala compiler.

All existing quasiquotation systems I know have been tied to a specific abstract syntax tree representation of the manipulated programs, which is limiting. Indeed, it is often useful to manipulate programs expressed in more advanced intermediate representations, as we will see in Chapter 4. In this situation, it was previously necessary to abandon the usage of quasiquotes and drop down to explicit manipulations of the underlying data structures of the intermediate representation, a difficult and error-prone task.

Thus, we show that Squid quasiquotes are *reusable*, in the sense that they are not tied to a particular program representation, but can be used to manipulate different intermediate representations. Adapting (or *binding*) a new IR to Squid is done simply by implementing a well-defined interface in the style of object algebras (or *tagless-final* style).

3.1 Introduction

In Chapter 1, we have detailed several different high-level approaches to manipulating program fragments, but all of these approaches assumed a more or less advanced *abstract syntax tree* representation of the programs.

On the other hand, the need often arises for more advanced IRs than plain ASTs [Stanier and Watson, 2013], such as ANF (Administrative Normal Form) [Flanagan et al., 1993], SSA (Static Single Assignment) or CFG (Control Flow Graph). This is particularly important when DSLs start incorporating effects and mutability, where evaluation order and aliasing become significant.

Manipulating this sort of intermediate representations is even harder and error-prone than manipulating plain abstract syntax trees (which was described in Section 1.2.1). One generally needs to propagate types manually as well as secondary meta-information like effects annotations, which one should have to deal with explicitly. IR transformation becomes very error-prone, especially since one has to be careful to account for effects and avoid performing transformations that would change the evaluation order of programs. The design of a deeply-embedded DSL and of associated program transformations (such as domain-specific optimizations) quickly becomes entangled with these low level IR implementation concerns, which get in the way of DSL designers.

In this chapter about the modular implementation of Squid, we first describe how Squid abstracts over the intermediate representation and provides general facilities to implement closed-world and open-world IR backends for the quasiquotes — in this sense, we say that Squid quasiquotes are “*generic*” in the IR. We are not aware of any previous generic quasiquote system. In the rest of the chapter, we describe various other concerns related to the implementation of the Squid system in Scala.

3.2 The Intermediate Representation Base

Figure 3.1 shows the Base trait required to be implemented by all Squid backends, taking the form of an *object algebra interface* [Oliveira and Cook, 2012] — also known as the *tagless-final* style [Carette et al., 2009] — where abstract type `Rep` represents the type of IR nodes, while types `Val` and `TypeRep` represent the types of bound variables and type representations, respectively.¹ Method `readVal` converts a variable symbol into a variable reference. `ascribe` corresponds to type ascription (of syntax `x : T` in Scala). Classes `Code` and `CodeType` have protected constructors in order to prevent external users from instantiating them arbitrarily.

Notice that Squid does not internally use a typed view of the IR (we have `Rep` instead of `Rep[T]`). This choice was motivated by simplicity of the Squid implementation and of the code generated by each quasiquote, ensuring faster compilation. Moreover, we noticed that when dealing with low-level IR manipulation, types often get in the way rather than help. Critically, this does not impact the soundness of high-level IR manipulation using quasiquotes, as high-level quasiquote terms are wrapped inside the typed `Code[T]` wrapper.

¹We do not show the methods for building type representations (`TypeRep`), but they follow the same pattern as for term representations (`Rep`).

```

trait Base {
  type Val
  type Rep
  type TypeRep

  def const(value: Any):          Rep
  def freshVal(name: String, typ: TypeRep): Val
  def readVal(v: Val):           Rep
  def lambda(param: Val, body: => Rep): Rep

  // override if needed:
  def ascribe(self: Rep, typ: TypeRep): Rep = self

  class Code[+T] protected(val rep: Rep)
  class CodeType[T] protected(val trep: TypeRep){ type Typ = T }

} // ...more helper methods and definitions elided

```

Figure 3.1 – Abstract types and methods required for a Squid IR.

3.3 Closed Worlds

Perhaps surprisingly, the Base trait does not feature a function application method. This is because in Scala and Squid, applying a function corresponds to calling the *apply* method defined on the `scala.Function` type, and Squid has a special mechanism for encoding methods in a user-extensible way: when generating IR code for a method call inside a quasiquote, Squid looks for a method with a corresponding name in the Base. If no such method is found, a compile-time error reports the missing feature. To avoid name clashes, these methods should live in objects whose names reflect the full names of the types where the original methods are defined. For example, to bind the IR in Figure 1.1a to a Squid base, we include the following definitions:

```

object MyIR extends Base {
  type Rep = Exp
  type TypeRep = Unit
  object `class scala.Int` {
    def typeRep = ()
    def + (self: Rep)(arg: Rep) = Add(self, arg)
  }
  object `class scala.Function` {
    def typeRep(lhs: TypeRep, rhs: TypeRep) = ()
    def apply(self: Rep)(arg: Rep) = Apply(self, arg)
  }
}

```

```
    }  
    /* ... more definitions ... */  
  }
```

Remark that in Scala, identifiers delimited with back-ticks may contain any valid characters, so we literally named the objects above “`class scala.Int`” and “`class scala.Function`”.

As an example, the code generated for `code" (x: Int) => x+1 "` after having imported the ‘`code`’ quasiquote builder from `MyIR` will be of the form:

```
val x = MyIR.freshVal("x", MyIR.`class scala.Int`.typeRep)  
val rep = MyIR.lambda(x,  
  MyIR.`class scala.Int`.+(MyIR.readVal(x), MyIR.const(1)))  
new MyIR.Code[Int](rep)
```

We do not give the full IR binding here for brevity. The online Squid repository contains several examples of custom Squid IRs, as well as a binding to an existing IR for the LMS-style DSL that was used in [Shaikhha et al., 2016].

3.4 Language Virtualization

For the economy of concepts, many Scala language features are internally encoded using the set of Base features that we have seen above. For this purpose, Squid defines a small library of *virtualized* constructs [Moors et al., 2012, Jovanovic et al., 2014]. For example: variables are represented using a `MutRef` data type supporting operations `!` and `:=` for variable access and modification respectively; if-then-else and loops are implemented using functions such as `ifThenElse` and `While` taking by-name arguments; by-name arguments themselves are represented as calls to a `byName` function taking a `() => T` function parameter; finally, functions with more than one parameter are implemented with curried functions passed into `uncurryN` methods — for example, `(x: Int, y: Int) => x+y` is represented as `uncurry2((x: Int) => (y: Int) => x+y)`; pattern matching is represented using `isInstanceOf` and `unapply` calls.² Finally, by default let-bindings are represented as lambda abstractions immediately applied (*redex*).

Naturally, these virtualized encodings can be safely ignored by quasiquote users, and DSL designers may convert them into their own IR-specific representations. For example, in:

```
object MyIR extends Base {  
  object `object squid.lib` {  
    def ifThenElse(cond: Rep, thn: Rep, els: Rep) =  
      buildInternalIfThenElseNode(cond, thn, els)  
  }  
  // ... more definitions elided  
}
```

²A more handy representation of pattern matching is left as future work.

3.5 Open Worlds

In the context of metaprogramming “at large,” like when writing general-purpose Scala macros (as opposed to DSL program transformations), it is useful to have a way to generate method applications on the fly, without having to define IR bindings manually for all possible methods.

This is possible thanks to the `OpenWorld` trait shown in Figure 3.2. If a base extends this trait, Squid will default to generating calls to `methodApp` to encode method applications that do not have a direct binding defined. `methodApp` takes a `tp` parameter so that the IR is informed of the type returned by the method call. `loadMtdSymbol` takes an overloading index to identify which method overload is being selected (0 if the method is not overloaded).

As an example, assuming we do not have in `MyIR` a direct binding for type `Double` and method `toDouble`, the quasiquote `code"2.toDouble"` will expand into the equivalent of:

```
val _Int = MyIR.loadTypSymbol("scala.Int")
val _Double = MyIR.loadTypSymbol("scala.Double")
val _toDouble = MyIR.loadMtdSymbol(_Int, "toDouble", 0)
val rep = MyIR.methodApp(MyIR.const(2),
                        _toDouble, Nil, Nil, typeApp(_Double, Nil))
new MyIR.Code[Double](rep)
```

The simplest way to implement methods `loadTypSymbol` and `loadMtdSymbol` is to make use of Scala Reflection’s runtime metaprogramming capabilities, reusing its `TypeSymbol` and `MethodSymbol` data types. This way, it is possible for an IR to dynamically explore things such as the annotations attached to a Scala method and its parameters, which is especially useful for implementing such mechanisms as annotation-based effect systems. Squid provides a ready-made `ScalaSymbols` trait that defines `loadTypSymbol` and `loadMtdSymbol` using Scala runtime reflection, so it is effortless for an IR to leverage these capabilities.

Finally, notice that using an open world IR generally allows for more flexibility. For example, it is possible to define programs that completely abstract over the base that is being used. Moreover, an open-world IR can be used as target to reinterpretation, as we will see in Section 3.7.

3.6 Support for IR Manipulation

In order to support pattern matching and term rewriting, a Squid IR has to extend yet another trait — `InspectableBase`, shown in Figure 3.3. The `Extract` type represents the result of pattern matching, and contains a mapping from term variable names to extracted terms and from type variable names to extracted type representations. `InspectableBase` defines the semantics of term and type pattern matching (`extract` and `extractTyp`), rewriting³ (`rewriteRep`),

³`rewriteRep` can be implemented in terms of `transform` and `extract`, but we found that for advanced IRs such as ANF, it is often useful to have more control on the way rewritings apply, enabling more powerful patterns.

```
trait OpenWorld extends Base {
  type MtdSymbol
  type TypSymbol

  def loadTypSymbol(fullName:String): TypSymbol
  def loadMtdSymbol
    (typ: TypSymbol, symName: String, index: Int): MtdSymbol
  def methodApp(self: Rep, mtd: MtdSymbol,
    targss:List[TypeRep],argss:List[ArgList], tp:TypeRep): Rep
  def typeApp(typ: TypSymbol, targss: List[TypeRep]): TypeRep

} // ...more helper methods and definitions elided
```

Figure 3.2 – The “open world” trait, which allows using any methods.

code traversal/transformation (transform), term equivalence (repEq) and subtyping (typLeq). Term equivalence is needed because Squid allows an extracted variable to be used in the same pattern, as in **case code** "(\$a, a) " which matches only pairs with twice the same component. Similar to ScalaSymbols for symbol loading, Squid provides a ready-made ScalaTyping trait that defines TypeRep, typLeq, typeHole and extractTyp relying on Scala’s runtime type representation facilities.

Pattern matching is implemented by building an IR node representing the pattern, where unquotes are replaced with special “hole” nodes. The IR then provides the semantics of matching a given *expression* node against that *pattern node*. Thus methods hole and typeHole represent unquotes in patterns. In addition with name and expected type, hole takes two lists of bound values yes and no, that specify respectively which bound references the hole is supposed to contain, and which it is forbidden to contain. For example, in pattern **case code** "(x:Int,y:Int,z:Int) => \$f(x,z) ", the argument to yes will be List(x,z) and that to no will be List(y). The hole method is supposed to extract a function term with arity equal to the length of yes. In the case above, it would extract an (Int, Int) => Int function term. On extraction, Squid lifts that term into a host-language function of type (Code[Int],Code[Int]) => Code[Int] automatically. Importantly, the term extracted by a hole may contain any references that appear in neither yes nor no. This is because code pattern-matching can be used in rewrite rules, which traverse every sub-term of a program and may open an arbitrary number of bindings on the way — as long as the terms extracted by a rule’s pattern end up as part of the rule’s result,⁴ these bindings should not be affected.

In contrast with Base and OpenWorld, providing an implementation for InspectableBase is usually a non-trivial undertaking, the most difficult task being to implement complete pattern matching semantics. On the other hand, once this is in place, one can fully benefit from

⁴And are not *extruded* by imperative effects like variable update.


```
trait InspectableBase extends Base {  
  type Extract = (Map[String, Rep], Map[String, TypeRep])  
  
  def extract (xtor: Rep, xtee: Rep): Option[Extract]  
  def rewriteRep(xtor: Rep, xtee: Rep,  
    mkCode: Extract => Option[Rep]): Option[Rep]  
  def extractTyp(xtor: TypeRep, xtee: TypeRep,  
    va: Variance): Option[Extract]  
  def transform(r: Rep)(pre: Rep=>Rep, post: Rep=>Rep): Rep  
  def hole(name: String, typ: TypeRep,  
    yes: List[Val], no: List[Val]): Rep  
  def typeHole(name: String): TypeRep  
  def reinterpret(r: Rep, newBase: OpenWorld): newBase.Rep  
  
  def repEq(a: Rep, b: Rep): Boolean  
  def typLeq(a: TypeRep, b: TypeRep): Boolean  
}
```

Figure 3.3 – Base for allowing code inspection (e.g., pattern matching).

Squid’s powerful and safe IR manipulation capabilities.

3.7 Intermediate Representation Reinterpretation

An important capability shown in Figure 3.3 is that offered by the `reinterpret` method: an `InspectableBase` may provide the capability to have its programs reinterpreted into a *different* Squid Base, which is an important tool that in turn enables many interesting applications (cf. Section 3.8). This is especially useful for optimizing high-level programs by progressively lowering their level of abstraction: at a certain point, we may want to switch to an IR which is more appropriate to deal with low-level programs.

Notice that `reinterpret` takes an `OpenWorld` parameter as the target Base, because it has to be able to reinterpret arbitrary features that may or may not be specially handled in the target IR. In practice, it is possible to adapt a non-`OpenWorld` IR to make it `OpenWorld`, using Java reflection to find the correct node creation methods at runtime.

3.8 One Interface to Rule Them All

...and in Abstraction Bind Them

In this subsection, we describe how Squid’s object algebra interfaces turned out to be a

powerful tool that facilitated the implementation of several Squid features.

Code generation backend. It can be useful to convert a program expressed in some custom IR into a standard Scala AST. This is simply done by *reinterpreting* that code into the `ScalaAST` base, in which `type Rep = universe.Tree` (where `universe.Tree` is the type of Scala ASTs). For example, in that base we have `def const(value: Any) = Literal(Constant(value))`, which constructs a Scala AST for a constant literal.

Note that IRs that rely on virtualized constructs [Moors et al., 2012, Jovanovic et al., 2014] will typically refine the behavior of the `reinterpret` method in case the target is a subclass of `ScalaAST`, so that these constructs are correctly de-virtualized. For example, without de-virtualization we might observe the following behavior:

```
scala> code"if (true) 1 else 0".reinterpretIn(new ScalaAST)
res0: universe.Tree = q"squid.lib.ifThenElse(true, 1, 0)"
```

To avoid this, the IR can special-case each virtualized construct in `reinterpret`, so that the expression above results in the expected Scala AST: `q"if (true) 1 else 0"`.

Pretty-printing. Pretty-printing is a standard application of object algebras [Oliveira and Cook, 2012], and requires defining an algebra where `type Rep = String`. However, when we already have an `InspectableBase`, we can avoid writing a pretty-printer entirely: it suffices to `reinterpret` the code into `ScalaAST` and then reuse the standard Scala pretty-printer.

Evaluation by runtime reflection. Squid provides the `ReflectInterpreterBase` implementation that leverages Java runtime reflection to execute code at runtime. In that base, we have `type Rep = Runner[Any]` (where `Runner` is a data type that is used to build a runnable representation of the code), and `methodApp` uses Java reflection to load the correct method from its method symbol and create the appropriate runner. Thanks to this interpreter, running code from an arbitrary `InspectableBase` is as simple as calling `reinterpret` with a `ReflectInterpreter` instance — in fact, Squid provides a `run: T` helper method on `Code[T]` types that does just that.

Evaluation by runtime compilation. A much more efficient but heavyweight way to implement code evaluation is to rely on Scala’s runtime compilation capabilities. We can use the Scala compiler to generate extremely efficient JVM byte-code at runtime, a useful capability for performance-sensitive systems that rely on staging.

Modular embedding. Remember that Squid leverages the Scala compiler to type check snippets of code, and then uses the result to build the corresponding IR nodes. We call our approach “*modular embedding*,” because the IR construction process itself is abstracted, and is expressed in terms of the `OpenWorld` interface. For example, the case that lifts constants from the type-checked Scala AST is of the form:

```
case Literal(Constant(x)) => base.const(x)
```

Where `base` is the `OpenWorldBase` object used to build the result of the embedding. The call to `const` refers to the function declared in Figure 3.1. This approach has the advantage that we can use modular embedding in different contexts:

- In the `optimize{...}` block construct shown in Chapter 4: the `optimize` macro embeds a piece of code at compile time into a given Squid IR where optimizations are performed, then reinterprets the code into the `ScalaAST` base to produce the result of the macro expansion. A similar mechanism is used in the code generated by the `@squidMacro` construct presented in Section 1.6.
- In quasiquotes, which embed code snippets into a specific `MirrorBase` backend, whose role is to generate the Scala AST necessary to reconstruct the same code at runtime. In this base, `const(42)` results in the Scala AST `q"$base.const(42)"`, where `base` identifies the target runtime base. Indeed, the role of quasiquotes is to create *run-time* code representations, as opposed to `optimize` whose goal is to handle code representations *at compile time*. Interestingly, the code invoked by `optimize` itself makes use of quasiquote-based “runtime” code manipulation — indeed, the runtime of the optimizer is the compile-time of the user program.

3.9 Implementation of Squid Quasiquotes in Scala

This section is aimed at giving the reader a better understanding of the mechanisms underlying Squid, as well as giving prospective implementers of advanced type system techniques insights on how Scala facilitates such endeavors.

The main takeaway is that the combination of a flexible type system with an advanced type-aware macro system can go a long way towards implementing advanced statically-typed features without modifying the host language’s compiler, provided that compiler supports macros with the capabilities listed in Sections 3.9.3 and 3.9.4.

3.9.1 Compilation of Squid Quasiquotes

Squid quasiquotes are implemented as macros that perform parsing and type-checking of quoted fragments, compute and check associated types and contexts, and produce the Scala code necessary to reconstruct the program fragments at runtime encoded in Squid’s intermediate representation.

Basic Expansion. To understand how Squid quasiquotes are compiled, let us start with a simple example, `code"Math.pow($x, 2)"`, where some value `x` is in scope with type `Code[Int]`. In Scala, this expression is conceptually equivalent to a simple invocation of the form `code(List("Math.pow(", ", 2"), x)`. The code function being a macro, it executes *at compile-time*. The first thing it does is to interpret the strings passed in its first argument as a Scala code

snippet. To do this, it reconstitutes the fragment as `"Math.pow(hole[Int](0), 2)"` and parses it using the Scala parser. `hole[Int](0)` represents the unquoted value `x`, where type argument `Int` was retrieved from its type in the current scope, and `0` is a unique identifier associated to the unquote. This snippet of code is then type checked using Scala's type checker, given signature `def hole[T]: T`. In this case, we end up with `"java.lang.Math.pow(hole[Int](0).toDouble, 2.0)"`, typed `Double`. Notice the insertion of `toDouble` as a result of type checking: similarly, the Scala type checker adds missing type parameters, inferred implicit arguments, fully-qualified names, etc. The next step is to *lift* this typed AST into a program that reconstructs it at runtime. During this process, `hole[Int](0)` is replaced with `x.rep`, where method `rep` accesses the underlying implementation of a code fragment. We give below a simplified version of the code that is produced:

```
val Math = staticObject("java.lang.Math")
val Math_pow = methodSymbol("java.lang.Math.pow")
val res = methodApp(Math, Math_pow, x.rep, Constant(2.0))
new Code[Double](rep = res)
```

Where `class Code[+T](rep: Rep)` is a typed wrapper that hides its internal untyped code representation `rep`. The expansion of quasiquotes in pattern position is very similar, desugaring to a call to the `extract` method that takes a pattern AST, a scrutinee AST and produces either nothing if the matching failed, or a mapping from extracted term names to extracted code fragments and extracted type names to extracted type representations.

Type-Parametric Matching. Type-parametric matching (see Section 1.3.4) uses the ability of Scala to reason about path-dependent types, which are types that may live in arbitrary objects, including local ones (this is a similar concept to first-class modules in ML). Squid assigns to an extracted type representation `ty` the type `CodeType` which contains an abstract type member `T` (a type declaration without a definition). Then, Squid type checks the pattern using references to `ty.T`. In essence, `ty.T` — which can only be referred to within the scope of the pattern matching branch where `ty` is extracted — is existentially quantified, which is the correct interpretation of type-parametric matchings. For example, assuming `pgrm` has type `Code[Any]`, in the code below (where `==~` stands for α equivalence):

```
pgrm match {
  case code"List[$ty]($a, $b)" =>
    print(ty) // ty is a term here
    val ls = code"List($b, $a)"
    assert(ls ==~ code"List[$ty] ($b, $a)")
    assert(ls ==~ code"List[ty.T]($b, $a)")
    ls
}
```

the pattern is type checked as having type `List[ty.T]`, and therefore the right-hand side of the match sees a scope with extracted variables `{ty: CodeType; a: Code[ty.T]; b: Code[ty.T]}`.

The return type of this example is `Code[List[_]]` — the wildcard in `List[_]` stands for an unknown type (an existential without a path). This is because the local type representation module `ty` is invisible from outside the scope of pattern matching branch, which ensures that extracted types from different patterns or even from different runs of a match cannot be mixed with one another.

3.9.2 Cross-Quotation References

An important feature of a flexible quasiquotation system is the ability to manipulate open terms.⁵ Since Squid quasiquotes are type-checked and hygienic, a program fragment like `code"x + 1"` is not valid on its own, as `x` is not defined.

However, within a context where `x` is bound at *the same quotation depth* as its reference, `code"x + 1"` becomes a valid expression. For instance, `code"(x: Int) => ${bar(code"x + 1")}"` is a valid quasiquote (of type `Code[Int => Int]`): when evaluated, the inner quasiquote will embed a reference to the outer `x`, be processed by `bar`, and then the result (which will presumably still contain references to `x`) will be inserted into the outer quote, which binds `x`.

A syntax white lie

Technically, the syntax I have described above, which is the syntax I have been and will be using throughout my thesis, *does not actually work* as is.

How could it not? Let us try it:

```
scala> code"(x: Int) => ${ identity(code"x + 1") }"
```

```
<console>:15: error: Embedding Error: Quoted expression does not type check:
    not found: value x
```

This error is due to a limitation in the implementation of Scala macros: the arguments passed to macros always expand before the macro itself. Therefore, the outer `code"(x: Int) => ${ ... }"` macro does not have an opportunity to set up its binding context before `code"x + 1"` expands, so the latter fails with a type error.

Thankfully, there are many ways to work around this limitation — as we will see below. The greatest tragedy of my PhD is that none of those workarounds is truly satisfactory, being either downright inelegant or looking uncomfortably asymmetric with the pattern syntax.

⁵A capability notably missing from the earlier Scala Reflection statically-typed quotation approach, the *reify* macro (see Section 1.7).

Redemption

The most tolerable workaround is to drop the usage of actual *quotes*, and rely on braces instead of the string-literal syntax, which has the effect of letting the Scala compiler understand that the outer quotation is setting up a binding context:

```
// Using braces:  
code{(x: T) => ${ identity(code{x + 1}) }}}
```

This works, by invoking an alternative frontend to the Squid macros, but has some serious drawbacks.⁶ First, it means that we have to restrict the syntax allowed in quasiquotes to Scala's syntax exclusively. This is in contrast with the string-literal syntax, which support some useful meta-syntax that is not valid Scala. For example, the string-literal syntax supports quasiquotes which define polymorphic function literals with a concise syntax, as in `code"[T] => (x: T) => (x, x)"` although polymorphic function literals are not yet supported in Scala (in the current version 2). Another example is the insertion of first-class variables (which we see later in Chapter 6) into quasiquotes, as in `code"val $v = 0; $v + 1"` — the brace-based syntax does not work well for this. Second, the syntax highlighting in integrated development environments will not color quoted code in a different color, contrary to the quoted version (it is colored differently here through obscure LaTeX invocations). Last but not least, this syntax is asymmetric with quasiquote patterns, which cannot use braces (because of another limitation of Scala macros) and therefore *have* to use the a string-literal syntax.

Brace and string-literal syntaxes can even be mixed, but that looks even less regular:

```
// Mixing braces and quotes:  
code{(x: T) => ${ identity(code"x + 1") }}}
```

An alternative workaround is to escape the quotation of the inner quasiquote, as well as the *anti*-quotation of the outer one:

```
// Escaping the unquote and inner quote:  
code" "" (x:T) => $$ { id(code"x") } " "" "
```

Escaping anti-quotations is done by doubling the dollar sign \$\$ so that the Scala compiler does not consider it like an actual escape (which would prompt it to try and type check the escaped code before the outer quote expands). Escaping inner quote is done by tripling the quotation marks on the outer one, and is necessary because the Scala compiler (which is now blind to the anti-quote) would consider the opening " as closing the outer quote. Together, these allow us to process the outer quote first, establishing the appropriate context before expanding the inner one.

A major drawback of this approach is that we lose syntax highlighting in the escaped code. Using triple quotes and double dollar signs is also fairly verbose.

⁶The braces syntax also has advantages, such as supporting click-to-definition in IDEs.

Note that escaping anti-quotations is useful in other contexts too. For instance, Squid supports escaping anti-quotations to *insert* values into *patterns*, as in:

```
// match a println call on the specific code value in c0:  
case code"println($c0)" => ...
```

Finally, we can also use automatic function lifting (presented in Section 1.3.9) to achieve the same effect as cross-quotation references, but it is even more verbose than other alternatives:

```
// Using automatic function lifting:  
code"(x:T) => ${ (y: Code[T]) => id(y) }(x)"
```

In practice

As a result of these limitations, I noticed that many Squid users have resolved to using braces almost exclusively in expression code, resorting to quotes only for patterns,

3.9.3 Required Properties of the Macro System

In order to achieve its goals of static safety, the Squid quasiquote system relies essentially on two features of the host language's macro system:

- The ability to query type information and invoke the type checker during macro expansion: it should be possible to query the type of unquoted expressions in order to properly type check the quote. Additionally but not essentially, Squid accesses the type of the scrutinee in pattern matching (cf. Section 6.2.1) and type-checks a quoted program fragment in the same scope as the quote itself, which is why we can write `{import Math.pow; code"pow($x,2)" }`.
- The ability to refine the type of expanded macros: since both the type and the context requirements of program fragments are computed during macro expansion, it must be possible for the compiler to expand a macro call before knowing its final type, and use the precise type of the expansion to type check the rest of the program.

3.9.4 Use of Runtime Reflection and Metaprogramming

Implementation of run and compile. Method `run` is implemented using Java reflection to load the classes mentioned in the program fragment and execute their methods. An alternative method `compile` invokes the Scala compiler *at runtime* to produce efficient JVM bytecode from a program fragment, and then execute it without any interpretative penalty. This enables Multi-Staged Programming [Taha and Sheard, 1997] (MSP), a form of explicit partial evaluation. In MSP, the original program generates a program at runtime (first stage), which may in turn

generate new programs (second stage, etc.), each time removing computations that are known at the current stage, so that an efficient implementation is finally synthesized that executes faster than its unstaged counterpart.

Subtype Checking in Code Pattern Matching. Squid makes use of Scala runtime type representations, that it packages with the program fragments. This is because subtyping checks are performed at runtime to guide pattern matching on those fragments.⁷ Thanks to Scala’s reflection features, we perform subtyping checks at runtime, leveraging Scala’s advanced type system almost for free. For example, pattern `case code "$ls: Seq[AnyVal]"` should match `code"List(1,2,3)"` because `List[Int] <: Seq[AnyVal]`, but should not match something like `code"List(4.toString)"` or it would lead to inconsistencies in reconstructed programs (cf. `String <: AnyVal`). Note the necessity to annotate holes for which Scala cannot locally infer a type, like for `$ls` in the pattern example above. In contrast, pattern `code"Math.pow($x,$y)"` is fine because `Math.pow` is not overloaded nor polymorphic and only works with arguments of type `Double`.

3.10 Related Work

3.10.1 Quasiquotes for Domain-Specific Languages

Quasiquotes in MetaML [Sheard et al., 1999], Haskell [Najd et al., 2016], F# [Syme, 2006] and others were used to facilitate the implementation of embedded DSLs such as language-integrated queries [Cheney et al., 2013]. Earlier approaches such as LINQ [Meijer et al., 2006] also provided some level of language-integrated domain-specific program reification. [Najd et al., 2016] use TTH to build DSL programs for their alternative embedding of Feldspar [Axelsson et al., 2010], an approach they call *Quoted DSLs* (QDSL). In this approach, a particular DSL is implemented using the quasiquotation abilities of a host language, which requires significant heavy lifting behind the scenes (for example, retrieving type information [Najd et al., 2016]). Najd et al. propose that “Rather than building a special-purpose tool for each QDSL, it should be possible to design a single tool for each host language.” With Squid, we realized this vision for Scala: we presented a quasiquote-based metaprogramming framework that simplifies the deep embedding of DSLs and the design of associated program transformations.

The practice of deeply embedding DSLs in host languages, exemplified by the polymorphic embedding approach [Hofer et al., 2008], requires to encode each DSL feature in the host language as a special data type. This translates into a lot of boilerplate, especially when associated with the burden of defining a suitable interface for DSL users, and it reduces the flexibility of the DSL design and implementation process. In contrast, we propose a system where quasiquotes are used both as the front-end for DSL users and the tool used by DSL developers to describe their domain-specific optimizations. This means DSL designers can

⁷Note that type information needs only be associated with program fragments, and not with current-stage values, which means we introduce no runtime overhead for normal computations not involving metaprogramming.

immediately use the shallow interface of their DSL (i.e., defined as a simple library in the host language), and apply custom analyses and rewritings on it without the need for a dedicated deep representation.

4 Optimizing High-Level Libraries with Quoted Staged Rewriting

Multi-stage programming (MSP, or just *staging*) is a popular technique for programmatically removing code abstractions, thereby allowing for faster program execution while retaining modular high-level interfaces.

Unfortunately, techniques based on MSP suffer from a number of problems — ranging from practicalities to fundamental limitations — which have prevented their widespread adoption. MSP requires both the designers of an optimized library and the users of that library to adapt their code to the technique. This results in exposing users to metaprogramming constructs or in having to hide such constructs behind cumbersome interfaces which are time-consuming to develop — either approach resulting in worse user experience. Moreover, libraries developed using MSP are often hard to extend and to compose together .

In this chapter, we introduce *quoted staged rewriting* (QSR), an approach to defining optimizations as rewrite rules using statically-typed analytic quasiquotes backed by a normalizing intermediate representation (IR).

The QSR approach is “staged” in two ways: first, rewrite rules can execute arbitrary code during pattern matching and code reconstruction, leveraging the power and flexibility of MSP; second, library designers can orchestrate the application of successive rewriting phases or *stages*.

The advantages of using quasiquote-based rewriting are that: 1. library designers who wish to implement optimizations never have to deal directly with the normalizing intermediate representation hidden by the quasiquotes; and 2. that it allows for the definition of non-intrusive optimizations — in contrast with MSP, it is not necessary to adapt the entire library and user programs to accommodate optimizations.

We show how Squid’s modular design, which was described in Chapter 3, enables QSR and renders library-defined optimizations more practical than ever before: it allows library designers to write safe and powerful domain-specific optimizers that library users invoke transparently on delimited portions of their code base.

4.1 Introduction

We begin by providing some necessary background before presenting our quoted staged rewriting approach. This section also serves as a presentation of the related work.

4.1.1 Staging and Extensible Compilers

In Section 2.6.1, we succinctly introduced the broad principles of multi-stage programming (MSP, or just *staging*) [Taha and Sheard, 1997]. In this section, we get into some more detail on the technique, its history and applications, and its limitations.

MSP lets programmers syntactically distinguish multiple stages of execution in their programs. At each intermediate stage the program computes away what is known at this stage, and generates a new *residual* program meant to execute the next stage. The ultimate stage performs the task of the unstaged program, but in a more efficient way. MSP can be viewed as a form of partial evaluation with explicit annotations for binding-time analysis [Jones et al., 1993], or as a way to define type-safe program generators that work by iteratively composing code fragments together. MSP can be applied to both run time [Taha and Sheard, 2000, Taha, 2004] and compile-time [Ganz et al., 2001, Yallop and White, 2015] code generation. In the latter case, programs are made of two stages where the first stage is executed at compile time and the second stage corresponds to the final, compiled program.

A major limitation of MSP is that it generally offers no type-safe facilities to analyse code (it is *purely generative*), which greatly restricts its capabilities in terms of program optimization. Moreover, staging a library exposes users of that library to staging annotations,¹ which leak through its interface. Perhaps more importantly, staging requires to decide from the beginning which parts of the program are static (meant to be executed at program generation time) and which parts are dynamic, then building everything around that dichotomy, making it hard to evolve the design later on without extensive refactorings.

Still, staging has been successfully applied to optimizing domain specific languages (DSL), especially Embedded DSLs (EDSL) [Hudak, 1996] which are DSLs that are defined within a more expressive *host language* such as Haskell [Axelsson et al., 2010, Najd et al., 2016, Hudak, 1996] or Scala [Rompf and Odersky, 2010, Lee et al., 2011, Ofenbeck et al., 2013, Scherr and Chiba, 2015]. In this context, staging has been used to facilitate the definition of extensible compilers for performance-oriented DSLs and heterogeneous target platforms [Lee et al., 2011, DeVito et al., 2013, Puschel et al., 2005, Ofenbeck et al., 2013, Axelsson et al., 2010].

Generally speaking, these compilers reuse the frontend capabilities of their host (syntax and type system) but they convert programs into their own domain-specific intermediate representation (IR) before stringifying low-level code. This limits their ability to interact with

¹Type-Based Embedding eschews staging quotations [Rompf and Odersky, 2010], but requires more complex types, which also degrades the library interface [Jovanovic et al., 2014, Rompf, 2016].

code outside of the DSL. For example, a compiler for a streams DSL (see the LMS embedding of [Kiselyov et al., 2017]) by default can only handle primitive types, strings, arrays, functions, loops and tuples, and adding support for using other constructs (such as, for example, `BigInt`) requires extending the compiler’s IR, which involves significant amounts of boilerplate. Moreover, expressing non-trivial optimizations in these frameworks is hard and error-prone, as one has to deal with details of the IR with limited support for code pattern matching. Tools have been proposed to generate some of the boilerplate automatically [Jovanovic et al., 2014, Sujeeth et al., 2013] and solutions were sketched to make code rewriting easier [Rompf, 2016], but the fundamental limitations and intrinsic complexity of these approaches are still there, and the burden they impose on library users only partly lifted.

In the words of Cohen et al. [2006], “*[MSP] does not relieve the programmer from reimplementing the main generator parts for each target application. The only way to improve code reuse in the generator is to base its design on a custom intermediate representation, which may be almost as convoluted for application programmers as designing their own compiler.*”

4.1.2 User-Defined Rewriting

The idea of building program optimizations via high-level rewrite rules is far from new [Visser, 2002, Visser et al., 1998, Peyton Jones et al., 2001, Steuwer et al., 2015, Puschel et al., 2005, Farmer, 2015, Sloane, 2011, Visser, 2001, Klint et al., 2009, de Moor and Sittampalam, 1999]. However, few approaches have offered a lightweight, type-safe, language-integrated way of expressing these rules, as most rely on distinct specification metalanguages or complex code transformation combinators.

A notable exception, the Glasgow Haskell Compiler (GHC) [Peyton Jones et al., 2001] allows library writers to describe simple algebraic rewrite rules consisting of two expressions: a pattern, and a template to replace the pattern with when the rule fires. GHC tries to apply as many of these rules as possible as it performs its own optimization passes. There are no termination or correctness guarantees associated with the rewrite rules, as their objective is to let users make — at their own risk — domain-specific assumptions that the compiler cannot make.

There are two major limitations to this approach. First, while the rules are sufficient to express a variety of optimizations, they are limited in the patterns that they can match and in the code that they can generate. For example, it is easy to define a rule to rewrite $\text{pow } x \ 2$ into $x * x$, but the generalization of that rule to rewrite $\text{pow } x \ n$, where n is constant, into $x * \dots * x$ is not expressible. Second, library designers have very weak guarantees about the actual application of their rules when a program is compiled. The result is intimately dependent on the inlining behavior of GHC (which is affected by separate compilation), so that expert knowledge about the inner workings of the GHC optimizer is often required to achieve satisfying results [Peyton Jones et al., 2001].² As a consequence, the practice is to annotate functions with

²The GHC wiki has this informal bit about the behavior of rewrite rules in the context of list fusion: “Q: *Why*

INLINE or NOINLINE directives that sometimes need to refer to GHC’s own internal optimization phase numbers. Moreover, approaches like stream fusion — a popular deforestation technique [Coutts et al., 2007, Coutts, 2011] — have been shown to require more powerful rewriting facilities than simple GHC rewrite rules. This has sparked interest in HERMIT [Farmer et al., 2014, Farmer, 2015], an interactive system based on rewriting combinators that is significantly more complex. Older systems like MAG [de Moor and Sittampalam, 1999, 2001] have proposed language-integrated rewriting for specific purposes such as mechanized fusion, but with weak or no type preservation guarantees.

4.1.3 Quoted Staged Rewriting

In summary, staging is powerful but imposes a burden on both library users and library designers. Moreover, purely-generative staging disallows code inspection, which is limiting, and approaches that allow code inspection do so by exposing low-level IR constructs that are hard to manipulate. Rewrite rules in the style of GHC are easier to express and integrate more seamlessly with the host language, but they lack expressiveness and control.

In this chapter, we bring together the advantages of staging and rewriting into a unified framework, *Quoted Staged Rewriting* (QSR), based on Squid. We claim that our approach combines the flexibility and ease of use of language-integrated rewrite rules with the power and guarantees of static staging.

Our framework works by **user-defined, scoped optimizations**, whereby: 1. library designers express powerful domain-specific optimizations by way of type-safe quasiquote-based rewrite rules; and 2. library users write normal, unstaged code that they can surround with `optimize{ ... }` blocks in order to apply those library optimizations. With first-class control of inlining, users can abstract on the library’s constructs while letting the library see through these abstractions to apply its rewritings. Scoped optimizations are useful because it often makes sense to focus optimization efforts on the “hot execution paths” of a program, where we can afford to let the optimizer spend more time doing its job. In our experience, applying these aggressive optimizations to more code outside of the hot paths makes the general compilation slower but has rapidly diminishing results.

Rewrite rules, which are applied at compile time, are allowed to use **arbitrary computations**, a flexibility that places them on equal footing with staging. On the other hand, rewriting enables a more dynamic approach to binding-time analysis. Together with extensible pattern matching, this favors more modular optimization designs: rewritings with orthogonal concerns can be completely decoupled. Taking inspiration from transformation-based compilers [Rompf et al., 2013, Jones, 1996], where rewritings are interspersed with successive **lowering phases** that decrease the general level of abstraction, we allow optimization designers to specify at

does making one thing fuse sometimes make something else not fuse? A: Because the whole system is built around inlining, and no one really knows how to make that Do The Right Thing every time. Also, no one knows a better way to avoid basing it on inlining.”

which phase to inline which library abstractions. This is an essential feature to guarantee consistent abstraction removal and robust, predictable optimization — both staples of MSP. We make novel use of a simple IR and effect system, to soundly accommodate Scala's imperative features while enabling high-level algebraic rewritings. We thus reap the benefits of purity while still allowing the manipulation of low-level imperative programs.

4.2 Multi-Stage Programming Limitations Exemplified

In this section, we exemplify the limitations of traditional multi-stage programming approaches to designing optimizing libraries.

4.2.1 Staging the Power Function

The prototypical staging example is *power*, a function that raises a number x to the n^{th} power: since the exponent part n is often a statically known integer, it is tempting to specialize that function so that a call to it expands into a simple sequence of multiplications. Figure 4.1a presents the code for a *staged* power function, which takes a current-stage exponent n and returns a function from any Double code value base to a code value representing its multiplication n times. Triple quotation marks " " " introduce multi-line quotations.

The figure ends with a usage example for $n = 3$. The astute reader will notice that the unquoted expression `power(3)` has type `Code[Double] => Code[Double]`, whereas it is used as if it were of type `Code[Double => Double]`. The reason is that Squid automatically lifts any current-stage function `Code[A] => Code[B]` into a next-stage function `Code[A => B]` upon insertion (as explained in Section 1.3.9).

4.2.2 New Optimization Opportunity

When removing abstractions programmatically and performing aggressive inlining, optimizable patterns often emerge, including patterns that a programmer would never write explicitly [Peyton Jones et al., 2001]. For instance consider a simulation application that needs to compute the gravitational force between several different celestial bodies. Those of us who remember our physics course will point out that the relevant equation has the form:

$$F = G \frac{m_0 \cdot m_1}{d(p_0, p_1)^2}$$

which corresponds to the program of Figure 4.2. When the call to `distance` is inlined, the body of `gravityForce` ends up containing a call to `pow(sqrt(...), 2)`, which is clearly an inefficient identity:

```
G * plan0.mass * plan1.mass /
  pow(sqrt( pow(plan0.pos.x - plan1.pos.x, 2)
```

```
def power(exp: Int)(base: Code[Double]): Code[Double] =
  if (exp == 0) code"1.0"
  else {
    assert(exp > 0)
    if (exp % 2 == 0) code" "
      val tmp = ${power(exp/2)(base)}
      tmp * tmp
    " "
    else code"$base * ${power(exp-1)(base)}" }

val pow3 = code"${power(3)}".run
```

(a) Defining a staged power function.

```
import Math.pow // pow: (Double, Double) => Double

@bottomUp @fixedPoint
val powOpt = rewrite {
  case code"pow($base, 0)" => code"1.0"
  case code"pow($base, ${Const(exp)})"
    if exp.isWhole && exp > 0 =>
      if (exp % 2 == 0) code" "
        val tmp = pow($base, ${Const(exp/2)})
        tmp * tmp
      " "
      else code"$base * pow($base, ${Const(exp-1)})"
}
def pow3(x: Double) = powOpt.optimize { pow(x,3) }
```

(b) Rewriting the standard `Math.pow` function.

Figure 4.1 – Two approaches to optimizing the power function.


```
+ pow(plan0.pos.y - plan1.pos.y, 2) ), 2)
```

Naturally, we would like being able to optimize such patterns.

4.2.3 Limitations of Staging

Let us consider what happens if we stage the function of Figure 4.2, making use of the power function defined in Figure 4.1a. Assuming purely-generative staging like in MetaOCaml [Taha, 2004], there is no easy way to extend that definition of power to perform the “power-of-power” optimization described above. The staged function can no longer accept a `Code[Double]` as the base, since purely generative approaches do not allow *inspecting* or *decomposing* code fragments – only *creating* and *composing* them together. However, we need to know whether a given piece of code has the form of a square root to be able to eliminate a square operation performed on it.

Solving this issue typically involves defining an auxiliary data structure for code being constructed, that carries additional information about its underlying structure. Figure 4.4 shows a generalized definition `genPower` of the power function, that uses an algebraic data type `CodeRep` to retain information about the code: subclass `Pow` describes a piece of code that results from an application of the power function, while `Simple` corresponds to other code. Method `toCode` is used to *reify* that intermediate representation into a proper code fragment to be inserted into a quasiquote.

Notice how that change affected the way we define `pow3`. More complex usages of power have to change in an even more significant way, as is shown in Figure 4.3 where we adapt the planet simulation code seen previously to our new staging scheme. As one can see, both the implementation of `genPower` and its usage in `pow3`, `distance` and `gravityForce` become tremendously more complicated. We believe that this is why purely-generative staging is often relegated to the back end — i.e., used merely for end-of-the-pipeline code generation, while the front end of the library is defined in the finally-tagless style [Carette et al., 2009] and mostly hides staging.

Extensible compiler techniques obviate the need to explicitly wrap and unwrap code, by making the equivalent of `Pow` directly extend (inherit from) the compiler’s internal IR node type [Hofer et al., 2008]. Moreover, type-inference-based techniques help to hide staging annotations to some extent [Rompf and Odersky, 2010], which can be further improved by language virtualization [Moors et al., 2012, Jovanovic et al., 2014], but the added complexity and fundamental limitations are still there: DSL designers have to write the entire library with staging in mind, define IR nodes for all constructs meant to be supported by the DSL, and interact directly with details of the compiler’s IR (especially when defining rewritings [Rompf, 2016]).

```
import Math.{pow, sqrt}

val G = 6.67E-11

def gravityForce(plan0: Planet, plan1: Planet) =
  G * plan0.mass * plan1.mass /
    pow(distance(plan0.pos, plan1.pos), 2)

def distance(p0: Position, p1: Position) =
  sqrt(pow(p0.x - p1.x, 2) + pow(p0.y - p1.y, 2))
```

Figure 4.2 – Example simulation code using sqrt and pow.

```
def gravityForce(p0: Code[Planet], p1: Code[Planet]) =
  code"G * $p0.mass * $p1.mass / ${
    genPower(distance(code"$p0.pos", code"$p1.pos"), 2.0).toCode }"

def distance(p0: Code[Position], p1: Code[Position]) =
  sqrt(Simple(code" "
    ${ genPower(Simple(code"$p0.x - $p1.x"), 2.0).toCode }
    + ${ genPower(Simple(code"$p0.y - $p1.y"), 2.0).toCode }
    " " ")))

def sqrt(x: CodeRep[Double]) = genPower(x, 0.5)
```

Figure 4.3 – Simulation code adapted for the (new) staged interface.

```

abstract class CodeRep[T] { def toCode: Code[T] }
case class Simple[T](toCode: Code[T]) extends CodeRep[T]
case class Pow(cde: Code[Double], exp: Double) extends CodeRep[Double] {
  def toCode =
    if (exp.isWhole) power(exp.toInt)(cde)
    else code"Math.pow($cde,{Const(exp)})"
}

def genPower(base: CodeRep[Double], exp: Double) = base match {
  case Simple(c) => Pow(c, exp)
  case Pow(c,e) if exp.isWhole => Pow(c,exp * e)
  case _ => Pow(base.toCode, exp)
}

val pow3 =
  code"(x: Double) => ${ genPower(Simple(code"x"), 3).toCode }".run

```

Figure 4.4 – New definition of the staged power function, with support for optimizing the “power-of-power” pattern.

4.3 Quoted Staged Rewriting

Remember that Squid supports *pattern matching* on code fragments, an innovation over classical MSP languages. The syntax is **case** **code**"*pattern*"=> result. In a **code** pattern, the semantics of unquotes is no longer to *insert* but rather to *extract* code fragments found in the place where they occur. For example, the following program evaluates to **code**"2":

```
code"2 + 1" match { case code"($n: Int) + 1" => n }
```

4.3.1 Rewriting Math.pow

Figure 4.1b presents a rewriting that optimizes calls to `Math.pow` with integer exponents using binary exponentiation. `Const` is the constructor/extractor for constant values; for example `Const(2)` is equivalent to **code**"2" in both expressions and patterns. A rewriting is registered using a **rewrite** block containing pattern matching clauses. Each rewriting can be configured to apply in bottom-up or top-down traversal order, and can be made to apply repeatedly until a fixed point is reached. In this example we use bottom-up order and fixed-point recursion. Method `isWhole` simply tests whether a `Double` value is a whole number. Note that Squid uses an IR based on the A-Normal Form (ANF) [Flanagan et al., 1993], which means that non-trivial sub-expressions are let-bound, so that it is not a problem to duplicate the base argument extracted from the patterns. For example, `pow(readInt, 3)` is rewritten into:

```

val x_0 = readInt
x_0 * 1.0 * (x_0 * 1.0 * (x_0 * 1.0))

```

One can immediately notice several differences with the staged version. First we do not need to create a new, distinct power construct; instead we operate directly on Java's standard `Math.pow` method. This means that any programs using `Math.pow` can already benefit from our optimization without any changes to their business logic. In other words, QSR allows us to work directly on program representations instead of staged structures, but without having to define our own domain-specific IR. Moreover, the optimization of `pow3` in Figure 4.1b happens at compile-time which makes the user experience similar to built-in optimizations.

4.3.2 Extending the Rewriting

Implementing the “power-of-power” optimization by rewriting is straightforward, as we can simply add the following rewrite rules³ to those of Figure 4.1b:

```
case code" sqrt($x) "
  => code" pow($x, 0.5) "
case code" pow(pow($base, ${Const(a)}), ${Const(b)}) "
  if b.isWhole
    => code" pow($base, ${Const(a * b)}) "
case code" pow($x, 1) "
  => x // just for aesthetics
```

We can now wrap the body of `gravityForce` in Figure 4.2 inside a `powOpt.optimize{...}` block, which rewrites it into:

```
val x_0 = plan0.pos.x - plan1.pos.x
val x_2 = plan0.pos.y - plan1.pos.y
G * plan0.mass * plan1.mass / (x_0 * x_0 + x_2 * x_2)
```

There is one caveat however: the additional rules have to be inserted *at the beginning* of the `case` clauses of Figure 4.1b, otherwise an expression such as `pow(pow(x, 0.5), 2)` will be rewritten to `val tmp = pow(y, 0.5); tmp * tmp` by the second rule of Figure 4.1b, before the new rules can be applied, missing that optimization opportunity.⁴ This shows that the ordering of rewritings should be carefully considered by library designers. More generally, it is often useful to organize rewrite rules into separate phases. For example, considering that the implementation of `sqrt` is faster than that of more general-purpose `pow`, it would pay off to have a later phase that converts code of the form `pow(x, 0.5)` *back* into `sqrt(x)` before emitting the final code. The question of rewriting phases is exemplified further in Section 5.4.

³We require the outer exponent `b` to be a whole number to avoid performing unsound reductions, like `sqrt(pow(x, 2))` to `x` instead of `abs(x)`.

⁴The cases of a rewriting are tried in the order they are defined, similar to classical pattern matching (the `match` keyword).

```

@online
val powOpt = rewrite {
  case code"pow($base, ${Const(exp)})"
    if exp.isWhole && exp > 0
    => power(exp.toInt)(base)
  case code"pow($base, $exp)"
    => throw StagingError(
      "Non-static exponent: " + exp.show)
}

```

Figure 4.5 – Hybrid approach: rewrite rules that falls back to staged function and emit error on rewrite failure.

4.3.3 Hybrid Approaches and Online Rewriting

It is possible to freely combine rewriting and traditional staging. For instance, while fixed point rewriting is often useful, its use in Figure 4.1b could be considered overkill; instead of looping through the fixed point of the rewrite rule, we could just as well call a staged function that performs the looping itself,⁵ as demonstrated in Figure 4.5. In that configuration, the role of rewrite rules is to automatically extract static parts from unannotated programs, similar to binding time analysis (BTA). Reminiscent of online partial evaluation [Jones et al., 1993], the `@online` annotation specifies that a rewriting should be performed on the fly, as program representations are constructed. Online rewrite rules can be used to achieve a form of normalization: by restricting the space of representable programs, they make transformations simpler to express. Additionally, they can alleviate phase ordering problems [Rompf et al., 2013].

4.3.4 Guarantees and Control

Thanks to arbitrary code execution in rewrite rules and control over inlining (see Section 5.4), we make the argument that QSR is as powerful as compile-time staging. In particular, it preserves all the necessary control required by library designers, who wish to guarantee to library users that program optimizations apply reliably: if some rewriting could not be applied because static information could not be extracted, it is always possible for the rewrite system to emit a compile-time error and fail code generation, as is done in the second rule of Figure 4.5. Other valid behaviors in this case may be: emitting a compile-time warning but going through with code generation; simply logging the failure in a report that users can inspect in order to understand how to speed up their program; or doing nothing at all – which is what traditional compiler optimizations have been doing.

⁵Yet another hybrid approach would be to use code pattern-matching inside of a staged definition, lifting the purely-generative restriction.

An even stronger argument can be made following [Cheney et al., 2013] and [Najd et al., 2016], who rely on the subformula principle of normal proofs adapted to programming [Wadler, 2015] to guarantee that types that do not appear as subformulas of the types of the inputs and outputs of a program will be completely removed after sufficient normalization. For example, a program of type `Int => Int` that is internally defined using some `Stream` data type can be rewritten to a program that does not make use of `Stream`, as long as we can inline all `Stream` functions. The possibility of inlining the relevant functions is an integral part of the subformula principle: if we do not have access to the function body (and therefore cannot inline it), the function itself should be counted as part of the inputs to the program, which prevents the application of the subformula principle (as the function type will contain the offending type — here, `Stream`).

4.3.5 Modularity of Rewritings

Squid enables the common approach [Visser et al., 1998, Sloane, 2011] of separating term-level rewritings from transformation strategies. As a result, it is possible to define self-contained libraries of useful rewritings as well as libraries of useful transformation strategies and compose them modularly. To combine different strategies we rely on Scala's mix-in composition mechanism, a technique also used in previous work [Hofer et al., 2008, Rompf et al., 2011].

Another important direction for modularity is to allow abstracting over patterns in rewrite rules [Visser, 2001]. Squid achieve this by merely relying on Scala's built-in custom extractors:

```
object Even {
  def unapply(x: Code[Double]): Option[Code[Double]] = x match {
    case Const(n) if n % 2 == 0 => Some(x)
    case code"($_: Int) * 2" => Some(x)
    case code"${Odd(_)} + 1" => Some(x)
    case _ => None
  }
}

object Odd { /* similar definition elided */ }

rewrite {
  case code"pow(-1, ${Even(n)})"
    => code"1.0"
  case code"pow(pow($b, ${Even(Const(n))}), ${Const(e)})"
    if n * e == 1.0
    => code"abs($b)"
}
```

The code above defines co-recursive `Even` and `Odd` extractors that are used to rewrite terms

such as `pow(pow(x, 2), 0.5)` into `abs(x)` and `pow(-1, readInt*2)` into `readInt; 1.0`.

4.3.6 Composing Uses of QSR Libraries

Finally, we describe how to compose together optimizers defined in different libraries. The simplest way to achieve composition is to *nest* the `optimize` blocks, which expand inside-out: in `r0.optimize{ ... r1.optimize{ ... } ... }` the `r1` block expands first (applying its rewrite rules), and what the `r0` block sees is the resulting optimized code. Consequently, this approach may yield different results depending on the order in which the different blocks are nested.

A more fine-grained alternative is to merge rewriting passes together, as in `(r0+r1).optimize{ ... }` but this requires that the rewritings be defined using compatible traversal strategies. More advanced library optimizers (like in Chapter 5) may be defined in terms of successive rewriting and inlining phases; determining how to mix these more complex optimizers together in a fine-grained way requires careful consideration from the user.

4.3.7 Optimizing Existing Libraries

As we saw with `Math.pow` and `Math.sqrt`, Squid can optimize code written using preexisting, unmodified libraries. On the other hand, it is often beneficial to design libraries with optimization in mind, using constructs that can be easily manipulated and optimized by code rewriting. For example, Chapter 5 presents a streams implementation `Strm` that is geared towards QSR. Nevertheless, it is still possible to use that ad-hoc implementation to optimize existing libraries, such as Scala's `Stream`. This is done in three phases. First, we define conversion functions `toStrm` and `toStream` to move between the two representations [Ureche et al., 2015]. Then we rewrite all `Stream` operations to `Strm` ones using these conversions, while collapsing useless conversions on the fly. For example, we convert `Stream.from(0, 1).take(3).sum` to `Strm.from(0, 1).take(3).sum` with these rewritings:

```
case code"($xs: Stream[Int]).sum"
  => code"toStrm($xs).sum"
case code"Stream.from($start, $step)"
  => code"toStream(Strm.from($start, $step))"
case code"($xs: Stream[$t]).take($n)"
  => code"toStream(toStrm($xs).take($n))"
case code"toStrm(toStream($xs: Strm[$t]))"
  => xs
```

Finally, the usual `Strm` optimizations can be applied on the resulting program, producing optimized code that may entirely bypass the usage of `Stream`. Note that in certain cases, inserting back-and-forth conversions may be *detrimental* to performance when the whole

pipeline cannot be converted and when the cost of conversion outweighs the gains of optimization [Coutts et al., 2007]. Thankfully, it is easy to write a separate “clean-up” phase which reverts conversions that could not apply fully, avoiding unwanted conversion costs.

4.4 Enabling Quoted Staged Rewriting

In this section, we detail several important technical aspects of the Squid implementation that enable QSR.

4.4.1 Effect-Sensitive A-Normal Form (ANF)

Hash consing. The Squid ANF IR is geared towards making rewrite rules as flexible as possible. As such, we have an unconventional definition of “non-trivial expressions” (those expressions that need to be let-bound). In our approach all pure expressions are considered trivial and therefore they are never let-bound. Semantically, it is *as if* pure expressions were duplicated at every one of their use sites, but internally Squid uses hash-consing [Jerding et al., 1997] so that there is only a single representation in memory of every pure term. Conceptually, writing `code<println(x+1); x+1>` is equivalent to writing `val x_0 = code<x+1>; code<println($x_0); $x_0>`. This is not only useful to save memory, but also allows Squid to cache transformations so that they are not performed more than once on a given pure term.

Matching. The mechanism described above gives us the benefits of ANF (normalized control-flow, sound handling of effects) while enabling powerful code pattern matching, because patterns can freely inspect nested sub-expressions as long as these sub-expressions are pure. Impure patterns can also be used, such as `case code<println(readInt)>`, which matches `code<val x = readInt; println(x)>` but does not match any program where there are impure expressions intervening between the `readInt` and `println` calls, like `code<val x = readInt; readDouble; println(x)>`.

Scheduling. When generating or pretty-printing code, Squid uses a *scheduling* phase to let-bind pure expressions that are used multiple times, in order to minimize program size and computation costs. This phase needs special care around closures, by-name arguments and branching constructs. For example, it makes sense to schedule expressions *out* of a loop (so as not to recompute them on every iteration), but *inside* an if-then-else branch if the other branch does not also use it (so as not to perform useless computations). Due to the lack of space, we do not describe the algorithm used by Squid to perform scheduling, but there is extensive literature on the subject [Click, 1995]. Squid allows users to annotate higher-order method parameters to indicate whether they are expected to execute *at most once*, *at least once* or *many times*. This way Squid can produce sensible schedules for code that uses custom constructs, such as the `loopWhile` function of Section 5.3.

4.4.2 Effect System

Basic Principles. In order to determine which expressions are pure, Squid uses a very simple yet surprisingly versatile effect system. The idea is to differentiate two kinds of effects: *direct* and *latent*. An expression has direct effect if it reads or writes mutable state, performs I/O, etc. Latent effects are reserved for expressions that *delay* the execution of direct effects, such as a lambda expression containing effectful code. Similarly to previous systems [Rytz et al., 2012], methods are then annotated with 1. their intrinsic effect, and 2. the way they propagate the effects of their arguments. “Pure expressions” are those with no direct effect, so lambda expressions are considered pure even when they have latent effect.

Examples. Since the Scala Stream datatype is purely functional, none of its methods has any intrinsic effects. However, transformers like `map` “build up” latent effects when applied to effectful functions, so `Stream(1,2,3).map(_ + readInt)` has latent effect. On the other hand, consumers like `fold` “execute” the latent effect of their arguments. This is because Stream is a *lazy* data structure that executes computations only when required. As an example, if either `s` or `f` have latent effect then `s.fold(0)(f)` has direct effect — otherwise it is pure. As a result, a stream pipeline like the ones we study in Chapter 5 is normalized to one big expression terminated by a call to a consumer such as `fold` or `foreach`, which allows for simple yet powerful rewritings (cf. Figure 5.2). Finally, notice that *strict* collections behave differently: for them, transformers execute immediately, so code like `List(1,2,3).map(_ + readInt)` has direct effect.

Future Work. Improving the effect system to be more fine-grained could benefit pattern matching and scheduling. In particular, we could maintain an arbitrary number of latent effect layers — currently we consider that `x => print(x)` and `x => y => print(x)` have the same (latent) effect, which means that when the latter is applied *once* it is already considered to have a direct effect, which is not unsound but rather imprecise. We could also use a graph-based IR [Click and Cooper, 1995] to maintain explicit dependencies between expressions, like in LMS [Rompf et al., 2013] or Graal [Würthinger, 2011]. Finally, while we currently require users to annotate the effects of their methods, automatic effect inference is entirely feasible. Existing dedicated effect-tracking tools could also be leveraged, such as Scala FX [Rytz et al., 2012].

4.4.3 Scalability of Code Pattern Matching

Squid implements pattern matching by building a term containing holes to represent the pattern, similarly to the *Folds* subsystem of HERMIT described in Farmer’s dissertation (p.73) [Farmer, 2015].

Squid’s current rewriting algorithm works by trying each pattern one after the other, and does not memoize previous matching results, although that would be possible to implement — following Farmer, we plan to use trie maps in order to speed up the process. Nevertheless,

we have found that even our non-optimal approach was practical, and enabled advanced rewritings like those of Chapter 5 plus dozens of online normalization rules⁶ to be applied on mid-sized method bodies without incurring concerning compilation times. We reserve an empirical study of these performance characteristics for future work.

⁶When we did the microbenchmarks of Section 5.8, there were a total of 66 online rewrite rules registered, handling things ranging from logic operations to options normalization to desugaring common Scala idioms.

5 Application: A New Approach to Stream Fusion

Stream fusion is a technique to automatically remove the creation of intermediate lists in functional programs.

Previous approaches to stream fusion have been based on simple rewrite rules, which suffered from brittleness and a lack of optimization guarantees, and multi-stage programming (MSP), which suffered from the limitations of MSP seen in the previous chapter.

In this chapter, we describe a solution based on quoted-staged rewriting (QSR) which reaps the advantages of both approaches, and also performs *more* fusion than either in some cases, thanks to the use of an advanced `flatMap-streamlining` program transformation.

Our implementation of stream fusion (a well-known deforestation technique) is both simpler and more powerful than the state of the art, and can readily be used by Scala programmers with no knowledge of metaprogramming.

5.1 Previous Approaches

The goal of *deforestation* is to optimize libraries which make use of functional data structures by removing the computation of intermediate results [Wadler, 1988]. List and stream fusion are two particular well-known deforestation techniques which deal with functional lists — or “streams” of data. We will henceforth refer to these techniques simply as *fusion*.

Fusion has been an intense subject of research [Gill et al., 1993, Gill, 1996, Peyton Jones et al., 2001, Coutts et al., 2007, Jonnalagedda and Stucki, 2015a, Kiselyov et al., 2017]. Promising approaches relying on simple Haskell rewrite rules [Peyton Jones et al., 2001, Coutts et al., 2007] were thoroughly explored, but these approaches often suffer from a lack of control and from limitations of the rewrite rules framework.

Multi-stage programming has been used to achieve some of the goals of fusion [Jonnalagedda and Stucki, 2015a, Kiselyov et al., 2017]. Most recently, Kiselyov et al. [Kiselyov et al., 2017] demonstrated an approach based on staging that fuses several difficult stream operations,

including “zipping” two streams together.¹ Their approach requires an elaborate staged representation of streaming code, that relies on existentially-quantified types to encode the stream’s internal state and uses continuation-passing style (CPS) thoroughly to thread state and iteration code through the streaming constructs. This makes the description and implementation of the library slightly convoluted and hard to understand. On the other hand, the library exposes staging annotations, as it forces users to write all the business logic of their application inside quotations.² For example, instead of writing `stream.map(x => x.foo)`, one has to write the equivalent of `stream.map(x => code "$x.foo")`. This has two disadvantages: first, library users generally have to use a compiler modified for staging, and need to understand staging annotations even when it’s irrelevant for their business logic; second, this means the BTA of the library is completely fixed [Leißa et al., 2015]; i.e., which parts are known statically is fully determined in advance by the library designers. Future changes to enable more optimizations may break the library interface, as we describe further in Section 5.6, and any usages of the library in a slightly more dynamic setting are impossible.

In the rest of this chapter, we show how to use QSR and reap the benefits of staging and rewriting: we perform stream fusion without affecting the user interface of the library and more thoroughly than in the staging-based previous work, and we enable more control and more powerful transformations than offered by GHC rewrite rules.

5.2 Stream Fusion by CPS and Inlining

The streams interface we focus on is the same as in [Kiselyov et al., 2017]; we show it below (syntax `type Strm[A] <: { ... }` specifies, through a subtyping bound, which methods the `Strm` type should contain). All functions are standard and have self-describing signatures:

```
type Strm[A] <: {  
  def map[B](f: A => B): Strm[B]  
  def flatMap[B](f: A => Strm[B]): Strm[B]  
  def take(n: Int): Strm[A]  
  def filter(p: A => Bool): Strm[A]  
  def zipWith[B](that: Strm[B]): Strm[(A,B)]  
  def fold[B](z: B)(f: (B,A) => B): B  
}  
def fromRange(from: Int, until: Int): Strm[Int]  
def unfold[A,B](init:B)(next:B => Option[(A,B)]): Strm[A]
```

Other constructs can of course be defined on top of these ones by adding “syntactic sugar,” such as `fromArray` in:

¹Zippping a stream of elements of type A with a stream of elements of type B produces a stream of combined (A,B) elements.

²The authors propose to use combinators to hide staging and mitigate the issue, but we believe that this is not really helping. `x => stagedFoo(x)` is not qualitatively better for the user than the quoted version.

```
def fromArray[A](xs: Array[A]): Strm[A] =
  fromRange(0, xs.length).map(i => xs(i))
```

A simple way to implement streams is by backing them with imperative *producers* which allow requesting elements one by one (the *pull* model of iteration) while keeping internal iteration state. In order to retain the expected pure interface for streams, it is necessary that stream objects not store a specific producer, but rather a way to initialize a new producer and its internal state — a producer factory:

```
case class Strm[A](producer: () => Producer[A])
```

`Producer[A]` can be implemented as a function of no arguments that, when called, returns `Some(e)` if `e` is the next element to be produced, or `None` if there are no more elements to produce (i.e., `type Producer[A] = () => Option[A]`). However, as has been noted before [Taha, 1999] the use of `Option` to guide control flow tends to generate code that is not easily optimized or partially evaluated, because it typically contains redundant branching expressions. In general, rewriting these into a more streamlined control flow requires some control-flow analysis. While this can certainly be done using our approach (since we can inspect code by recursively traversing it via pattern-matching), it is much easier to adopt an alternative representation of producers. As often, the better representation is in continuation-passing style:

```
type Consumer[A] = A => Unit
type Producer[A] = Consumer[A] => Unit
```

Figure 5.1 shows the gist of the `Strm` implementation that we finally settle on. The `andThen` combinator pipelines two functions together such that `f.andThen(g)` (also written in operator syntax `f andThen g`) is equivalent to `x => g(f(x))`.

Marking the `Strm` class with an `@embed` annotation allows Squid to automatically create a deep embedding for the body of every method in the class (similar to [Jovanovic et al., 2014]). By default, unless annotated with an explicit `@phase` (see Section 5.4), methods and data constructors are treated by Squid like syntactic sugar, and they are inlined on the fly.

Perhaps surprisingly, most of the constructs of our streams library already fuse automatically after inlining. For example, the “*Hello World*” of fusion `xs.map(f).map(g).sum` where `s.sum = s.fold(0)(_ + _)` basically desugars/inlines into:

```
val p = xs.producer(); var cur = 0
var cont = true; while (cont) { cont = false
  p { a => cur = cur + g(f(a)); cont = true } }
cur
```

As a more interesting example, consider the program:

```
optimize{ (xs:Array[Int]) => unfold(0)(i => Some(i,i+1))
```

```
@embed
case class Strm[A](producer: () => Producer[A]) {

  def map[B](f: A => B): Strm[B] = Strm(() => {
    val p = producer()
    k => p(f andThen k)
  })
  def take(n: Int): Strm[A] = Strm(() => {
    val p = producer()
    var taken = 0
    k => if (taken < n) { taken += 1; p(k) }
  })
  def zip[B](that: Strm[B]): Strm[(A,B)] = Strm(() => {
    val p0 = producer()
    val p1 = that.producer()
    k => p0 { a => p1 { b => k((a,b)) } }
  })
  def fold[B](z: B)(f: (B,A) => B): B = {
    val p = producer()
    var cur = z
    var cont = true
    while (cont) {
      cont = false
      p { a => cur = f(cur, a); cont = true }
    }
    cur
  }
  def foreach(f: A=>Unit): Unit =
    fold(()) { (_, a) => f(a) }

  /* other implementations elided */
}

def fromRange(n: Int, m: Int): Strm[Int] = Strm(() => {
  var i = n
  k => { if (i < m) { k(i); i += 1 } }
})
```

Figure 5.1 – Implementation of the Strm data type.

```
.zip(fromArray(xs).filter(_ % 2 == 0)).foreach(print) }
```

For which Squid produces this code, slightly reformatted:

```
(xs_0: Array[Int]) => {
  val len_1 = xs_0.length
  var st_2 = 0
  var i_3 = 0
  var cont_4 = true
  while (cont_4) {
    cont_4 = false
    val x_5 = st_2
    st_2 = x_5 + 1
    var cont_6 = true
    while (cont_6) {
      cont_6 = false
      val iv_7 = i_3
      if (iv_7 < len_1) {
        val x_8 = xs_0(iv_7)
        if (x_8 % 2 == 0) {
          print((x_5,x_8))
          cont_4 = true
        } else cont_6 = true
        i_3 = iv_7 + 1
      }
    }
  }
}
```

As we can observe, all of the `Strm` abstractions have been removed and closures have disappeared, leaving behind a residual program made only of variables and loops. Normalization plays a major role in this regard: on the one hand, Squid relies on ANF (cf. Section 4.4.1) to streamline block structures and inline “one-shot” lambdas (lambdas applied only once [Peyton Jones et al., 2001]); on the other hand, user-defined online rewrite rules allow getting rid of intermediate abstractions — notice that the code above does not contain any `Option` despite the `unfold` interface making use of them. We do not describe such normalizations here for lack of space, but the ones that apply in this case transform `Some(x).isDefined` and `Some(x).get` into `true` and `x` respectively.

5.3 The Problem with flatMap

The only construct that does not play well with this state of affairs is `flatMap`, which is due to its intrinsic higher-order nature. To understand this, consider one of its possible implementations,

shown below:

```
def flatMap[B](f: A => Strm[B]): Strm[B] = Strm(()) => {
  val p = producer()
  var curBp = Option.empty[Producer[B]]
  k => {
    var consumed = false
    loopWhile {
      if (!curBp.isDefined)
        p { a => curBp = Some(f(a).producer()) }
      curBp.fold(false) { bs =>
        bs { b => k(b); consumed = true }
        if (!consumed) { curBp = None; true }
        else false
      }
    }
  }
}
```

Syntax `loopWhile{...}` is the same as `while({...}){...}`, and `opt.fold(d)(f)` applies function `f` on the value contained in option `opt` or returns `d` if `opt` is not defined. The implementation proceeds by storing the current producer of `B` elements in variable `curBp`. Whenever the current producer runs out of elements (variable `consumed` is not set to `true` after calling `bs`), we set `curBp` to the next producer, which is obtained by applying `f` on the next element `a` of `p`.

The problem is that `curBp` is a variable that stores a function, preventing its inlining (remember that type `Producer[B]` is an alias for `Consumer[B] => Unit`). Notice that each time the value of `curBp` is reset, it captures a *different* value of `a` that is *not* available outside of the continuation passed to `p`. Even if we know the body of `f`, we cannot naively inline it at its use site, in `bs{ b => ... }`, because we would no longer have access to that `a`. As was noted before [Coutts et al., 2007, Coutts, 2011, Farmer et al., 2014], these complications derive directly from the power and generality of `flatMap`. In the general case, for each element of the source stream, the function passed to `flatMap` could return streams of arbitrary shapes constructed at runtime, making it unfeasible for a compiler to fuse the code based solely on static information. However, in a significant proportion of stream programs used in practice (if not the vast majority), `flatMap` is used with more “well-behaved” functions, for example functions that always produce the same shape of streams for each source element (see Section 5.7). Furthermore, it is often possible to reorganize a program so that the result of any `flatMap` is consumed all at once (the “push-based” approach) instead of one element at a time, which allows us to avoid the inefficient pull-based implementation shown above. In the next section we explore that approach, and in Section 5.7 we describe a more general but more complex and slightly less efficient solution.

5.4 Enabling More Fusion by QSR

Our goal is to defer the inlining of `flatMap` and the other `Strm` operations so that we get a chance to rewrite stream usage patterns in a way that removes the need for pulling from `flatMap` results. To achieve this, we annotate all core `Strm` methods (those that are not syntactic sugar) with `@phase("Low")` to delay their inlining. We then take inspiration from Kiselyov et al. [Kiselyov et al., 2017], who leverage the fact that `flatMap` is no more problematic if we can consume its elements using internal iteration (push-based approach, cf. `foreach`) instead of external iteration (pull-based). We introduce the notion of *pullable* streams for streams that can be efficiently used with external iteration. To mark streams that are pullable, we use a dummy “marker” method `pull[A](as: Strm[A]): Strm[A]` also annotated with `@phase("Low")`, which simply returns its argument unchanged.³ We make `fromRange` and `unfold` syntactic sugar that wrap their body with a call to `pull`, since their implementations are pullable, and we define the propagation rules seen in the first part of Figure 5.2. These rules “float out” the `pull` wrapper as long as the outer operation is also pullable.⁴

The next step is to define rules that fold usages of the stream combinators in order to enable internal iteration. To simplify this step, we redefine `fold` and `foreach` in terms of a `doWhile` method that consumes the elements of a stream as long as its argument function returns `true`:

```
@phase("Low") def doWhile (f: A => Bool) = {
  val p = producer(); loopWhile {
    var cont = false; p { a => cont = f(a) }; cont }}

```

The *Folding* rules in the second part of Figure 5.2 then reduce stream combinators that are applied to `doWhile`. The last two rules of Figure 5.2 dispatch the implementation of `zip` depending on which of its two arguments is pullable. Similar to [Kiselyov et al., 2017], in this section we do not explicitly handle the case where neither is pullable. Function `doZip` is syntactic sugar for a specialized version of `doWhile`:

```
def doZip[A,B](s: Strm[A], p: Producer[B])(f: (A,B) => Bool) =
  s.doWhile{ a => var c = false; p { b => c = f(a,b) }; c }

```

Schematically, our optimizer is organized as follows:

- **Desugaring:** This is already done automatically by Squid; it concerns `fromArray`, `fold`, `foreach`, `doZip`, etc.
- **Flow:** bottom-up rewriting to propagate the `pull` information “down” the method chain — when possible — and to reduce consumed streams using internal iteration.
- **Lowering:** inlining of the `Strm` constructor, `pull`, `doWhile` and other core `Strm` methods to low-level code; removal of `Option` variables and other low-level optimizations.

³A common technique, also used by GHC developers. For example see <https://ghc.haskell.org/trac/ghc/wiki/OneShot> (accessed June 28 2017).

⁴Squid allows type-parametric matching (extracting types, as in Section 4.3.7). For simplicity, Figure 5.2 does not show the type extractions.

Example. Consider the following pipeline transformation:

```
// Source:
fromRange(0, n) zip (
  fromRange(0, m).map(i => fromRange(0, i)).flatMap(x => x)
) filter {x => x._1 %2 == 0} foreach println

// After Desugaring:
pullable(fromRangeImpl(0, n)).zip(
  pullable(fromRangeImpl(0, m))
    .map(i => pullable(fromRangeImpl(0, i)))
    .flatMap(x => x)
).filter { x => x._1 %2 == 0 }
  .doWhile { x => println(x); true }

// After Flow:
val p = fromRangeImpl(0, n).producer()
fromRangeImpl(0, m) doWhile { i =>
  var cont_0 = false
  fromRangeImpl(0, i) doWhile { b =>
    var cont_1 = false
    p { a => if (a %2 == 0) println((a, b)); cont_1 = true }
    cont_0 = cont_1
    cont_0
  }
  cont_0
}

// After Lowering, the code has only variables and loops
```

To conclude this part, let us remark that we already fuse more programs than [Kiselyov et al., 2017], because in contrast with that work we do not desugar `filter` to `flatMap`. The implementation of `filter` is pullable while that of `flatMap` is not, so that desugaring is counterproductive. As a result, we can fuse programs such as `(s0 filter f0)zip (s1 filter f1)` while [Kiselyov et al., 2017] cannot — in their case writing such a program results in the generation of variables holding closures that capture local mutable state, a failure of abstraction removal.

By using a careful design, simple high-level rewrite rules and controlled inlining, we achieved state-of-the-art stream fusion capabilities with less complexity than previous work.

```

@bottomUp @fixedPoint val Flow = rewrite {

  // Floating out pullable info
  case code"pull($as) map $f"
    => code"pull($as map $f)"
  case code"pull($as) filter $pred"
    => code"pull($as filter $pred)"
  case code"pull($as) take $n"
    => code"pull($as take $n)"
  case code"pull($as) flatMap $f"
    => code"$as flatMap $f" // flatMap is not 'pullable'

  // Folding
  case code"pull($as) doWhile $f"
    => code"$as doWhile $f"
  case code"$as map $f doWhile $g"
    => code"$as doWhile ($f andThen $g)"
  case code"$as filter $pred doWhile $f"
    => code"$as doWhile { a => !$pred(a) || $f(a) }"
  case code"$as take $n doWhile $f"
    => code" "
      var tk = 0
      $as doWhile { a => tk += 1; tk <= $n && $f(a) }
      " " "
  case code"$as flatMap $f doWhile $g"
    => code" "
      $as doWhile { a =>
        var c = false
        $f(a) doWhile {b => c = $g(b); c}
        c
      } " " "

  // Zipping
  case code"$as zip pull($bs) doWhile $f" => code" "<
    $as.doZip($bs.producer()){ (a,b) => $f((a,b)) } " " "
  case code"pull($as) zip $bs doWhile $f" => code" "<
    $bs.doZip($as.producer()){ (b,a) => $f((a,b)) } " " "

}

```

Figure 5.2 – Algebraic rewrite rules for stream fusion.

5.5 Correctness of the Stream Fusion Scheme

The first desirable property for our stream fusion rewriting system is that it terminates, expressed in Theorem 5.5.1 below:

Theorem 5.5.1 (Strong Normalization). *The fixed point application of the rewrite rules in Figure 5.2 always converges.*

Proof. The pull wrapper propagation converges because pull is only propagated outwards, and is never introduced by any other rule. For the rest of the rules, notice that they each reduce the number of `Strm` constructs in the program by at least one. This is a decreasing measure which ensures that the rewriting is terminating.

Next, let us argue that we fuse all stream pipelines that we set out to fuse (which excludes pipelines zipping two flattened streams). Interestingly, Figure 5.2 can be viewed like small-step operational semantics, where values are fully-fused stream pipeline programs. Our goal is then to show that well-formed pipelines reduce to values, which is done via the usual properties of *subject reduction* (Theorem 5.5.4) and *progress* (Theorem 5.5.6).

Definition 5.5.2 (Pullable Stream). *A stream term that can be rewritten to a term of the form `pull(xs)` by the rewrite rules in Figure 5.2.*

Definition 5.5.3 (Well-formed pipeline). *A well-typed program where: 1. all stream sub-terms are used as arguments in applications of `map`, `flatMap`, `filter`, `take`, `zip`, `pull`, or `doWhile`; 2. where any applications of `zip` has at least one of its two arguments pullable; and 3. where applications of `pull` enclose neither `zip` nor `flatMap` applications.*

Notice that in actual programs, `pull` is not invoked by the user but solely arises from desugaring `fromRange(n.m)` and `unfold(z)(f)` into well-formed sub-terms, respectively the terms `pull(fromRangeImpl(n, m))` and `pull(unfoldImpl(z)(f))`.

For simplicity we consider that `producer()` calls introduced by the *Zipping* rules are immediately inlined, and that the resulting code is inlined as well, recursively.

Theorem 5.5.4 (Subject Reduction). *The rewrite rules of Figure 5.2 preserve types and well-formedness.*

Proof. We have type preservation for free thanks to Squid, which statically enforces that the result of each rewrite rule has the same type as the pattern. It is straightforward to see that well-formedness is preserved as well, as the rules only introduce `Strm` functions and low-level constructs — we can prove by induction that `producer()` is never applied on `flatMap` (which is the only construct with a non-trivial producer implementation) because the zipping rules make sure `producer()` is only applied on pullable streams, and `flatMap` does not propagate the pull wrapper.

Definition 5.5.5 (Fully-fused pipeline). *A program that only contains references to `doWhile`, `fromRangeImpl`, `unfoldImpl` and low-level constructs such as variables of primitive types, if-then-else branches, conditionals, etc.*

Theorem 5.5.6 (Progress). *Any well-formed pipeline that is not fully-fused can have at least one of its sub-terms reduced by the rules of Figure 5.2.*

Proof. For a pipeline to be well-formed but not fully fused, it either needs to have non-low-level features such as function variables — which cannot happen because we only inline pullable streams — or it needs to still have one of `pull`, `map`, `flatMap`, `filter`, `take` or `zip`. At least one of these has to be passed into a call to `doWhile`, by well-formedness hypothesis (because `doWhile` is the only terminal operation). Therefore, such term can be reduced by one of the folding rewrite rules if the outer term is not a `zip`. If it is a `zip`, we can either propagate `pull` in one of its arguments, or we can apply one of the zipping rules because by hypothesis at least one of the two arguments is pullable.

Finally, remark that *semantic* preservation (in terms of program execution semantics) is easily assessed by looking at each rewrite rule case in isolation. In other words, QSR makes it easy to reason locally about each rewrite rule, ensuring that it transforms its input program into an equivalent output.

5.6 Extensibility of Optimizations

We already saw that Squid allows adding syntactic sugar in the form of user-defined methods annotated with or enclosed by a class annotated with `@embed` (which allows Squid to lift the method's implementation). Here we examine how to extend the set of core stream constructs.

As an example, consider stream programs that contain code of the form `if (...) stream0 else stream1`. At the moment, that pattern will not be recognized as pullable by the library, and may therefore get in the way of fusion. Thankfully, by only adding the two rules below we can seamlessly integrate that pattern with the rest of the fusion system:

```
// Floating out pullable info
case code"if ($c) pull($thn) else pull($els)"
=> code"pull(if ($c) $thn else $els)"

// Folding
case code"( if($c) $thn else $els ) doWhile $f"
=> code" " "
    if ($c) $thn doWhile inl($f)
    else    $els doWhile inl($f)
    " " "
```

Like `pull(s)`, syntax `inl(f)` is used as a marker. It hints for Squid to inline function `f`, even if `f` is used in several places. This effectively leads to code duplication in the case above, but that is a requirement for fusion to happen reliably.

Contrast the seamless extension above with what [Kiselyov et al., 2017] proposes to solve the same problem, which involves *changing the user interface* of `flatMap` to continuation-passing style.

5.7 When Everything Else Fails — Fusing `flatMap` the Hard Way

The rewriting proposed in Section 5.4 generates fused code for many use cases, but unfortunately fails to fuse `zip` applications where both arguments are flattened. More generally, it fails to fuse `flatMap` in the absence of a direct consumer of the flattened stream, which can also happen if we only have access to incomplete pipelines, such as `(n: Int) => fromRange(0, n).flatMap(fromRange(0, _))`.

In general, it would be useful if we could *streamline* `flatMap` applications so as to make them efficiently pullable. Intuitively, the code above could be rewritten:

```
(n: Int) => Strm(() => {  
  var i = 0  
  var j: Option[Int] = None  
  k => loopWhile { // loop until suitable element is found  
    if (j.isEmpty && i < n) { j = Some(i); i += 1 }  
    j.fold(false) { jv =>  
      if (jv < i) { k(jv); false } // element is found  
      else { j = None; true } // j stream is exhausted  
    }  
  }  
})
```

Notice the similarity with the implementation of `flatMap` shown in Section 5.3. The main difference is that instead of using a variable that stores the inner `Producer`, we use a variable that stores the *state* of the inner producer (here `j`), and that state is reset whenever a new value of the outer producer (here `i`) is obtained. The idea is similar to the one proposed in [Farmer et al., 2014], though we use actual imperative stream states while [Farmer et al., 2014] uses a purely functional encoding of stream states, as the host language (Haskell) is purely functional.

This transformation can be applied automatically, provided we have access to the complete inlined state of the inner producer. This tells us that the rewriting should apply after the *Lowering* phase of Section 5.4. To prevent `flatMap` from being inlined to its inefficient implementation, we change its implementation so that it inlines into a low-level `doFlatMap(p, f)` method where `p` is the source producer, and `f` is the function that creates an inner producer from each element of `p`. Next, we register the rewrite rule of Figure 5.3, to be applied during

Lowering. The code in Figure 5.3 is the most technical example of this chapter; to understand it, we first need to introduce a few concepts:

The defaultValue helper. The `defaultValue[T]` helper method, which can be used within quasiquotes allows one to create, for a given type `T`, a dummy “default” value on the current compilation platform, to be used as a placeholder in initialization positions while the real value has not yet been computed.

Higher-Order Patterns Variables. As explained in Section 1.3.10, Squid provides higher-order patterns variables, whereby, for example, pattern `code" (x: Int) => $body: Int"` will not match a lambda where body makes use of `x`, but the following pattern will: `code" (x: Int) => $f(x): Int"`, giving to `f` type `Code[Int] => Code[Int]`.

Temporary Variable Extrusion. In order to inspect open code, which is represented as a function [Pfenning and Elliott, 1988], we must apply it to some value first. However, sometimes we do not yet have that value until after we have inspected the code. To solve this, Squid provides an `open(f)((body, close) => ...)` idiom used to temporarily manipulate the open body of code function `f` as `body`, making it inspectable. Given some `f : Code[A] => Code[B]`, the type of `body` is `B` but it implicitly contains unbound references to its `A` parameter. `close` has type⁵ $\forall X. \text{Code}[X] \Rightarrow \text{Code}[A \Rightarrow X]$, and is used to reintroduce the explicit parameter dependency. `close` can be applied to `body` or to any of its subterms. This mechanism can lead to errors, if one does not `close` pieces of code that were closed and contain occurrences of the parameter. However, the `close` function checks that no such occurrences exist in the result. Therefore, any programmer error is immediately reported at the relevant place, and we never end up with programs containing unbound or wrongly-captured variables, which greatly simplifies debugging.

In Figure 5.3 we recursively analyse the body of `f`, using a `reset` parameter to accumulate a function term that resets the state of the inner stream when a new element of the outer stream is available. When we encounter a mutable variable binding, we reconstruct it but initialize it with `null` and integrate the actual initialization as part of the `reset` argument passed recursively. Immutable value bindings are converted into variables so they can be reset similarly. Finally, encountered effects are simply integrated into the `reset` accumulator. When the recursion finally encounters the `k => ...` lambda expression constructing the resulting `Producer`, we build the final, efficient implementation of `flatMap`. Note that it is important to match a lambda `k => ...` term, for the soundness of the rewriting (finding something else in place of a lambda would mean that low-level state inlining might not have been complete).

Summary. We presented an algorithmic rewrite rule that performs `flatMap` streamlining to enable fusion in many of the cases where it failed in Section 5.4. The rule is implemented entirely using the type-safe, high-level utilities provided by Squid. Previously, Farmer et al. [2014] demonstrated a similar rewriting, but to implement it they had to extend the compiler

⁵Scala 2 does not support first-class polymorphic function types (though they are being introduced in Scala 3), but such types can be easily emulated via a class type; if the class has an `apply` method, it will look the same as a function value from the user’s perspective.

```
@online val FlatMapStreamlining = rewrite {
  case code"doFlatMap[$ta, $tb]($pa, a => $f(a))" =>
    open(f) { (body_f, close_f) =>
      def rec(produce: Code[Producer[tb.T]], reset: Code[() => Unit])
        : Code[Producer[tb.T]] = produce match {
        // Adapt `var` bindings:
        case code"var x: $xt = $init; $innerBody(x, x = _)"
          => open(innerBody) { (inner, close_inner) =>
              code"var y = defaultValue[$xt]; ${
                val newReset = code"() => { $reset(); y = $init }"
                close_inner(rec(inner, newReset))(code"y", code"y = _")
              }"
            }
        // Adapt `val` bindings:
        case code"val x: $xt = $init; $innerBody(x)"
          => open(innerBody) { (inner, close_inner) =>
              code"var y = defaultValue[$xt]; ${
                val newReset = code"() => { $reset(); y = $init }"
                close_inner(rec(inner, newReset))(code"y")
              }"
            }
        // Adapt imperative effects:
        case code"$effect; $innerBody"
          => rec(innerBody, code"() => { $reset(); $effect }")
        // Conclude the rewriting by adapting the ending lambda:
        case code"k => $innerBody(k)" => code""
          var curA: Option[$ta] = None
          (k: Consumer[$tb]) => {
            var consumed = false
            loopWhile {
              if (!curA.isDefined)
                $p { a => curA = Some(a); ${close_f(reset)}(a)() }
              curA.fold(false) { a =>
                ${close_f(produce)}(a) { b => k(b); consumed = true }
                if (!consumed) { curA = None; true }
                else false
              }
            }
          }
        case _ => throw StagingError("Could not streamline this flatMap")
      }
    }
  rec(body_f, code"() => ()")
}
```

Figure 5.3 – The flatMap streamlining rewrite rule.

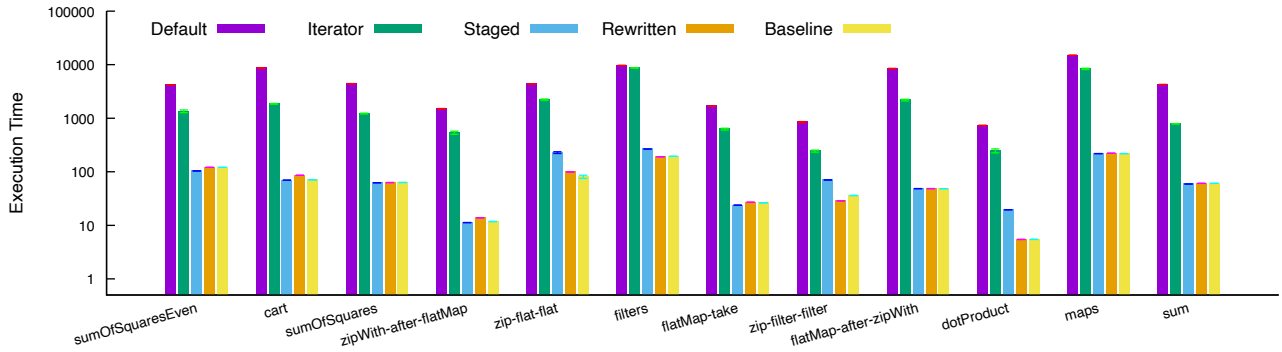


Figure 5.4 – Time taken by different stream pipeline implementations on the JVM. Notice the logarithmic scale. Default: our streams library without optimization; Iterator: standard Scala iterators; Staged: staged streams using LMS [Kiselyov et al., 2017]; Rewritten: our streams library with optimizations applied; Baseline: manual low-level implementations. The code is available online at <https://github.com/epfldata/staged-rewritten-streams>.

and drop down to complicated IR manipulations. We can also argue that our approach is more robust, as the user has full control on the rewriting and inlining pipeline. Remark that `flatMap` streamlining usefully completes the scheme of Section 5.4, but does not replace it: our optimizer always tries to apply the latter first, because it is more efficient (as it deals with smaller, higher-level stream representations) and because it tends to produce slightly better code with less variables and loops.

5.8 Evaluation

5.8.1 Performance

In this section, we empirically demonstrate that our QSR stream fusion approach⁶ is competitive with both staging and manual low-level implementations that use only integral variables and loops. We measured the execution time of small pipelines consisting of flat-mapping, filtering, zipping, etc. and summing up the results. We tested five approaches: our pure Scala library without optimization (Default); Scala iterators, which are conceptually similar but lower level (their interface is imperative) and hand-optimized to play well with the JVM (Iterator); the staged code of [Kiselyov et al., 2017] (Staged); the same code as Default but surrounded with an `optimize{...}` block to apply our QSR (Rewritten); and hand-coded low level implementations using integral variables and loops (Baseline). The inputs used consisted of arrays of several hundred thousand integer elements. The times were measured on a six-core Intel Xeon E5-2620 v2 processor with 256GB of DDR3 RAM (1600Mhz). We used Scala version 2.11.2 running on the OpenJDK 64-Bit Server VM (build 24.95-b01) with Java 1.7.0_101.

⁶The fusion algorithm we have benchmarked in this section is the one presented previously in this chapter but with a few minor tweaks and extra normalization rules that help with streamlining.

	Shallow	Fusion	Generic
Squid / QSR	127	149	293
LMS / [Kiselyov et al., 2017]	—	314	1982

Table 5.1 – Lines of code for stream fusion in Squid and LMS.

As we can observe, the rewritten version has performance characteristics mostly similar to the staged and low-level versions. All these three versions outperform the unoptimized and iterator versions by one or two orders of magnitude. We interpret that performance difference as the cost of abstraction (here mainly incurred from using closures, virtual dispatch and boxing). Even in these simple cases the advanced JIT of the JVM in server mode cannot remove that overhead automatically. Both the rewritten and staged approaches produce similarly low-level code where all abstractions have been eliminated, except for cases `zip_flat_flat` and `zip_filter_filter` where the staged version fails to fuse and falls back to using variables holding functions. The generated code still performs honorably — about twice as slow as the rewritten version but still an order of magnitude faster than the unoptimized ones. Notice that the rewritten version does completely fuse `zip_flat_flat` and `zip_filter_filter`, and therefore has comparable results with the baseline.⁷ For the rest of the tests, the minor differences in runtime between the staged, rewritten and baseline versions can be attributed to slight differences in generated looping structures.

5.8.2 Productivity

We conclude with a brief empirical argument about the productivity gains of our approach. We measured the number of physical lines of code (i.e., excluding comments and blank lines) in: 1. the “shallow” implementation of the library (cf. Figure 5.1); 2. the implementation of stream fusion; and 3. the supporting library code that allows the approaches to function (*Generic*). The LMS implementation does not have a shallow counterpart to its staged streams library — which is reported under *Fusion*. The *Generic* number for LMS accounts for the IR definitions of basic constructs such as arrays, strings, tuples, etc., and associated code generation implementations; it only includes LMS code used by this application. For Squid, *Generic* includes mainly standard normalization rules as well as a small library of virtualized constructs (like mutable variables). Notice that the stream fusion rewritings in Squid are half the size of the staging-based implementation in LMS, and are completely separate from the library, which can be used independently. Even more tellingly, the LMS approach requires considerably more supporting code,⁸ and that code will only grow as users want to include more constructs to be used within their streams programs. In contrast, Squid accommodates

⁷A previous version of these benchmarks [Parreaux et al., 2017a] had a visible difference between the rewritten and baseline versions on `zip_flat_flat`, but Oleg Kiselyov helpfully pointed out that our baseline implementation of the algorithm was wrong (we forgot to update some variables), which was responsible for the difference.

⁸This code can be generated automatically by tools like Yin-Yang [Jovanovic et al., 2014] and Forge [Sujeeth et al., 2013], but it is still an overhead compared to not needing it at all.

new constructs without requiring any additional supporting code.

5.9 Conclusion

Quoted Staged Rewriting allows the authors of high-level functional libraries to co-design associated domain-specific optimizers to improve the efficiency of their libraries. As demonstrated with the stream fusion use case studied in this chapter, this technique can even enable the same level of performance as manually-written low-level code.

6 Improved Safety and Expressivity for Analytic Metaprogramming

In Chapter 1, we saw how to statically type quasiquotes with analytic capabilities, unifying Lisp-style and MetaML-style quasiquotation.

However, the way we proposed handling open code was not fully satisfactory. First, it was not sound in the presence of effects like mutation and exceptions. Second, it was not sound in the presence of a `run` (or `compile`) function which interprets or compiles a piece of code on the fly. Third, open code extracted from patterns was encoded using higher-order patterns variables, which is very restrictive as it prevents freely inspecting extracted terms — indeed, it forced us, in Chapter 5, to use a *temporary variable extrusion* escape hatch, which when misused can violate static scope safety.

In this chapter, we describe a possible solution to ensuring the safety of open code manipulation. This new solution tracks *scope dependencies* in the types of terms and allows very flexible *non-lexically-scoped* open code manipulations. This lets us express scope-safe code generation in the presence of effects, temporary extrusion of bindings via code pattern matching, and the composition of open code fragments without syntactically-surrounding binders.

We formalize this approach as λ^{g} , a multi-stage calculus with code pattern matching and rewriting, and prove its soundness. We also present its realization in a new version of Squid (dubbed *Contextual Squid*), leveraging Scala’s expressive type system.

To demonstrate the usefulness of the approach, we introduce *speculative rewrite rules*, a novel code transformation technique which makes decisive use of these new non-lexical open code manipulation capabilities. This work was motivated by the needs of real-world metaprogramming applications; in Section 6.5 we see several examples of speculative rewrite rules inspired by query compilation use cases.

However, we will see in the next chapter that the approach presented in this chapter is in fact not completely adequate, as it does not support the more general forms of scope polymorphism. Therefore, this chapter should really be considered a stepping stone towards a more ambitious solution, and not a final say on the matter.

6.1 Introduction

In Chapter 1, we saw a version of *Squid* which supported hygienic code manipulations, in the sense that unrelated bindings would never interfere, but 1. it did not guarantee that all evaluated code was well-scoped; and 2. the primitives it offered to match and reconstruct bindings lacked expressiveness (see Section 6.6 for a more thorough comparison).

In this chapter, we describe and exemplify a more advanced version of *Squid*, referred to as *Contextual Squid* (though we may just use *Squid* from now on), which resolves these problems.

6.1.1 Motivating Example

As a motivating example for this chapter, let us consider the simple metaprogramming task of turning an array of tuples into a tuple of arrays.

For instance, given the following dummy source program:

```
(n: Int, idx: Int) => {  
  val arr = new Array[(Int, Int)](n)  
  var i = 0  
  while (i < n) {  
    arr(i) = (i, i * 2)  
    i += 1  
  }  
  arr(idx)  
}
```

the goal is to rewrite it into the following form:

```
(n: Int, idx: Int) => {  
  val arr_0 = new Array[Int](n)  
  val arr_1 = new Array[Int](n)  
  var i = 0  
  while (i < n) {  
    arr_0(i) = i  
    arr_1(i) = i * 2  
    i += 1  
  }  
  (arr_0(idx), arr_1(idx))  
}
```

This optimization is sometimes known as “array-of-structs to struct-of-arrays,” and has particular relevance in the field of databases (see Section 6.5.3); its goal is to streamline array

accesses, making them more cache-friendly for the processor, and to avoid the performance cost of allocating tuples.

6.1.2 Limitations of Higher-Order Abstract Syntax

This program transformation example is a good example of the limitations of strictly-lexically-scoped open code manipulation approaches, which we mentioned in Section 1.2.

The problem is that we have to match on usages of local array variables, and *change* in a non-trivial way the binding of that variable — in this case, we want to replace it with two distinct bindings, which goes beyond the simple substitution use cases where HOAS shines.

Let us try to see how we would write the transformation. We would start by matching an array variable declaration and its body:

```
pgrm rewrite {
  case code"val a = new Array[$t0]($size); $body(a): $t1"
    =>
    ...
}
```

From this pattern, we extract a size variable of type `Code[Int]` and a body variable of type `Code[Array[t0.T]] => Code[t1.T]`. Notice that body is a function and cannot be inspected directly. In order to transform the usages of the array variable inside the body, we need to somehow provide a value of type `Code[Array[t0.T]]`, but how shall we do that?

One solution would be to pass in a dummy program fragment for a temporary substitute of the original array variable, for example `code"dummy[Array[$t0]]({Const(freshCount)})"` where `dummy` and `freshCount` would be defined as follows:

```
def dummy[T](id: Int) = throw new Exception

private var curId = -1
def freshCount = { curId += 1; curId }
```

This essentially brings us back to the plain-AST approach of handling bindings (discussed in Section 1.2), with all the caveats that it entails: lack of scope safety and lack of hygiene — it is easy to misuse such constructs and end up with metaprograms mixing fresh counts and forgetting to make sure all dummy occurrences are properly removed from the manipulated programs. So it is not a very satisfactory answer.

6.1.3 Non-lexically-scoped Open Code Manipulation

Essentially, the metaprogramming ability we are looking for can be summarized as follows:

The ability to manipulate open terms (compose and decompose them) independently from the binders which will eventually capture the free variables contained in these open terms.

Because this implies the use of variables references in places which are not *lexically* (in the metaprograms) within the scope of their final binder, we refer to this as *non-lexically-scoped* open code manipulation. Another way of looking at the same problem is that in traditional metaprogramming approaches, it is often useful to separate the generation of variable names from their binding [Pottier, 2007].

This approach makes many use cases simpler, and allows us to extract concrete first-order *terms* from patterns containing bindings, which will let us express the array-of-tuples transformation mentioned above as well as the `flatMap` streamlining algorithm of Figure 5.3 much more safely and directly.

Moreover, this approach can let us delay decisions about the new bindings we want to introduce during program transformation (such as where or in what order to insert them, or whether to insert them at all) *after* having inspected the bodies of the original bindings.

However, in order to *statically ensure* that non-lexically-scoped metaprograms are still scope-safe and hygienic, we need a more advanced type system which is able to track the dependencies of program fragments in their types. The goal of this chapter is to demonstrate a system which does just that.

6.1.4 Early Example of Rewriting

Before we delve into the design of Squid's new type system, and to get a taste of the system, let us first review a much simplified version of our motivating example.

Consider the problem of finding all local variables that hold a pair of values (a, b) , removing these variables and rewriting their uses into direct accesses to a and b . The following Scala program uses Squid's `rewrite` primitive to traverse a `pgrm` term bottom up while matching any variable `p` bound to a pair of integers; in the scope `body` of each such binding, it replaces projections to `p`'s first and second components (syntaxes `p._1` and `p._2`) with the corresponding pair element a or b :

```
pgrm.rewrite {  
  case code"val p: (Int, Int) = ($a, $b); $body" =>  
    val body2 = body.rewrite {  
      case code"p._1" => a  
      case code"p._2" => b  
    }  
    body2.subst["p"] (code"($a, $b)")  
}
```


Contextual Squid statically keeps track of the fact that variable *p* is *free* in program fragments *body* and *body2* (i.e., these terms are “open in *p*”). Therefore, patterns used in the inner rewriting of *body* are allowed to refer to that variable *p*. Matching on the usage of existing free variables is a fundamentally important capability of our non-lexical system, which cannot be expressed directly using a higher-order approach like HOAS.

Expression `body2.subst["p"](c)` returns the *substitution* in *body2* of all occurrences of the variable named “*p*” with the provided code fragment *c*. This is used to replace all remaining occurrences of *p* (if any) with an in-place reconstruction of the original pair. For example, program `code"val my = (1, 2 + 2); print(my); my._1 + my._2"` is first rewritten into `body2 = code"print(my); 1 + (2 + 2)"`, and then into `code"print((1, 2 + 2)); 1 + (2 + 2)"`.

Contextual Squid is still hygienic, because it will not mix up the matched binding *p* with bound variables present in the original program *pgrm*, even if they also happened to be named *p* when the original program was constructed.

Moreover, Contextual Squid is scope-safe, because forgetting to substitute *p* in *body2* at the end of the rewriting will result in a type error, reported at compilation time — otherwise, our rewriting could result in programs with unbound references to *p*.

6.2 Presentation of Contextual Squid

We now review the features of Contextual Squid and see how to resolve the motivating example presented in Section 6.1.1.

6.2.1 Handling of Open Code in Contextual Squid

The way Squid allows the type-safe manipulation of non-lexically-scoped open program fragments is threefold.

Representing free variables explicitly

Squid allows expressing free variable references explicitly, without a lexically-enclosing binder. This is done by prepending a ‘?’ to the names of these free variables when they occur in a quasiquote.¹

The example given in Table 1.1, which uses a *cross-quotation reference*:

```
code"(x: Int) => ${ id(code"x + 1") }"
```

can now *also* be written in Squid as:

¹ We could avoid the ‘?’ and view all unqualified names as free variables (which is what is done in the formalism), but this would be a bad ergonomic choice: typos could easily result in confusing errors, and we do not want for e.g., `code"print(1)"` to be interpreted as `code"(?print)(1)"` instead of `code"scala.Predef.print(1)"`.

```
code" (x: Int) => ${ id(code" (?x: Int) + 1") }"
```

The main advantage is that it renders the composition of code more modular: an open code fragment can be written out without having bindings for all its free variables syntactically surrounding it; i.e., Squid quasiquotes do not have to be lexically scoped. In the previous example, we can extract the inner expression into a `val` binding, which does not work in approaches like MetaML:

```
val inner = id(code" (?x: Int) + 1")
code" (x: Int) => $inner"
```

Allowing violations of lexical scoping based on explicit annotations is related to the approach taken by Kim et al. [2006].

Reflecting context requirements in the type of code fragments

In order to track which fragments are open and when their free variables are to be captured, as well as to decide when code is safe to be evaluated, we make the types of quoted terms directly reflect their *context requirements*, i.e., the names and types of the free variables contained in those terms.

For instance, `val fx = code" (?x: Int) + 1"` has type `Code[Int, {x: Int}]`,² which means that it represents a term of type `Int` that needs to be inserted in some context where a variable `x` of type `Int` is defined. By composition, `code"$fx.toDouble"` has type `Code[Double, {x: Int}]` as it is equivalent to `code"((?x: Int) + 1).toDouble"`.

The free variables contained in a term that is inserted into some quotation context are correctly captured by the variables bound in said context: `code"val x = 0; $fx.toDouble"` has type `Code[Double, {}]` and evaluates to `code"val x = 0; (x + 1).toDouble"` (or, equivalently, `code"val y = 0; (y + 1).toDouble"`).

While Contextual Squid still allows the use of cross-quotation variable references, just like the original version of Squid, it type-checks them in a more careful way. When one writes code such as `code" (x: Int) => ${ id(code"x + 1") }"` (from Table 1.1), the type of the inner quote `code"x + 1"` is no longer just `Code[Int]`, but `Code[Int, {x: Int}]`.³

Allowing binding-destructuring patterns

Quasiquote patterns may be used to match bindings, and the extracted subterms will have types that reflect and track their potential dependencies to these bindings; e.g., using `→` to

²In Scala, `{val x: Int}` is a structural object type listing some immutable field `x` of type `Int`; for concision we omit the `val` and write it just as `{x: Int}` in this thesis.

³This is a good approximation for now, but is not quite exact. We will see in Section 7.4.2 that cross-quotation references in Squid are type checked in a more subtle way.

denote evaluation in a REPL session:

```
val f = code"(x: Int) => x + (?y:Int)"
      : Code[Int => Int, {y: Int}]
      → code"(x: Int) => x + ?y"

val g = f match { case code"(z => $body)" => body }
      : Code[Int, { y: Int; z: Int }]
      → code"?z + ?y"
```

Several things should be noted here.

First, `code` quasiquotes integrate well with Scala's type inference, as the type of the scrutinee `f` propagates to help type the pattern in `g`. If `f` had type `Code[Any, _]` instead and we still wanted to match an `Int => Int` function, we would have to write the pattern as `case code"(z: Int) => $body: Int"`.

Second, the names of *bound* variables do not matter, and a lambda that used `x` as the parameter name can be matched as if it were using `z` instead.

Third, in the example, the type of the extracted fragment `body` reflects that it may contain free variables from two different sources: by propagation from the scrutinee's type we know it may refer to some `y`, and because the pattern introduced a binding named `z` it may as well refer to it — what happened is that `body` was *safely extruded* from its enclosing context `{z: Int}`.

6.2.2 Rewrite Rules and Polymorphism

As we have seen in Section 1.3.2, Squid provides a `rewrite` method that traverses a program and applies a transformation while checking at compile-time that the transformation preserves the type and context of each sub-terms. Moreover, Squid allows the extraction of types along with terms.

Recall the example shown in Section 1.3.2: given some `pgrm` fragment we transform calls to `foldLeft` on `List` objects into imperative `foreach` loops:

```
pgrm rewrite {
  case code"($ls:List[$t]).foldLeft[$r]($init)($f)"
    => code"var cur = $init; $ls.foreach(x => cur = $f(cur, x)); cur"
}
```

For example, if `pgrm = code"List(1,2).foldLeft(0)((acc,x) => acc+x) + 1"`, the rewriting returns `code"{var cur = 0; List(1,2).foreach(x => cur = cur+x); cur} + 1"` (the β -redex is removed by Squid's internal normalization). Note that in Scala partial functions are written `{ case ... => ... }`; they are similar to pattern matching, but they need not be exhaustive.

Operator syntax ‘`p rewrite f`’ is the same as ‘`p.rewrite(f)`’. Notice that we extract a type `t` that is never used explicitly — it is in fact inferred as part of the type of the quoted program on the right-hand side of the rewriting case.⁴

6.2.3 Fixed Point Rewritings

Rewritings can be applied over and over again until they reach a fixed point. In Squid, rewriting `Math.pow(x, 2)` into `x * x` is trivially expressed, but let us here consider the generalization of that problem to arbitrary exponents.

We define below a fixed point rewriting that uses binary exponentiation to transform calls to `Math.pow` with a constant integer exponent into a series of multiplications. `Const` is the constructor for constants, used to lift current-stage values as code constants and extract constant values from code fragments. For instance, pattern `code"pow($x, ${Const(exp)})"` extracts `x` as a code value of type `Code[Double, _]`, but it extracts `exp` as a “bare” value of type `Double`. Method `isWhole` from class `Double` is used to query whether a floating-point number has an integral value.

```
import Math.pow

pgrm fix_rewrite {
  case code"pow($x, $exp)"
    if !x.isTrivial
    => code"val base = $x; pow(base, $exp)"
  case code"pow($x, 0)"
    => code"1.0"
  case code"pow($x, ${Const(exp)})"
    if exp.isWhole && exp > 0
    => if (exp % 2 == 0)
      code"val tmp = pow($x, ${Const(exp / 2)}); tmp * tmp"
    else
      code"$x * pow($x, ${Const(exp - 1)})"
}
```

The role of the first `case` rule, which is applied first, is to let-bind the base `x` passed to `pow` if it is not “trivial” i.e., unless it is a constant or a variable reference. This avoids code duplication that would otherwise result from the following rules.⁵ For example, `pow(.5, 3)` is rewritten into `0.5*{val tmp_0 = 0.5*1.0; tmp_0*tmp_0}`, duplicating `0.5`, but `pow(readDouble, 3)` (where `readDouble` reads a number from standard input) is rewritten into `val x_0 = readDouble;`

⁴ This named pattern variable is necessary (`$t` cannot be replaced with `$_`) because Squid needs to generate a local, named type symbol representing the extracted type. This is explained further in Section 3.9.1.

⁵ Note that Squid can be used with different underlying intermediate representations [Parreaux et al., 2017b]; by using an appropriate representation (such as the A-normal form), such code duplication concerns disappear, as we show in [2017a].

```
x_0*{val tmp_1= x_0*1.0; tmp_1*tmp_1}.
```

6.2.4 Free Variables and Substitution

As explained in the introduction, given some variable x free in t , we can replace all its occurrences in t with syntax $t.\text{subst}["x"](y)$, as in the following example:

```
val a = code"(?x: Int) + 1"
      : Code[Int, {x: Int}]
      → code"?x + 1"

val b = a.subst["x"] (code"27")
      : Code[Int, {}]
      → code"27 + 1"
```

Note that the right-hand side y of $t.\text{subst}["x"](y)$ is evaluated *lazily* (i.e., only if there are actual instances of x left in t), a property that proves useful in the next sections.

6.2.5 Speculative Rewrite Rules

In Squid, the current innermost rewriting can be aborted by calling `abort()` at any point in the right-hand side of the rewriting case. This call never returns and passes the control back to the rewriting engine.⁶

We call *speculative rewrite rule* a rewrite rule which attempts to apply a transformation optimistically, but aborts that transformation as soon as it finds something that prevents its successful application. In essence, speculative rewrite rules are a convenient and type-safe way to express conditional rewritings without having to define separate, error-prone analysis passes over the program one wants to transform.

In the next section, we present an example of speculative rewrite rule and explain how Squid ensures the safety of these constructs. Another example, directly extracted from our work on query compilation [Shaikhha et al., 2016], is given in Section 6.5.2.

6.2.6 Motivating Example: Array of Tuples Optimization

Figure 6.1 presents an example of speculative rewrite rule that attempts to turn any array of 2-tuple elements into two distinct arrays.⁷ A trace of the successive values taken by each variable, given a dummy input `pgrm`, is shown below:

⁶This mechanism is similar to delimited continuations [Danvy and Filinski, 1990], where ‘`case` pattern `=> ...`’ acts like `(reset (shift c ...))` and `abort()` acts like a short-circuiting `(c ())`. This is internally implemented on the JVM using exceptions.

⁷The JVM stores composite objects such as tuples using an additional level of indirection (boxing), which is removed if we store each field of the tuple in a separate array.

```
def optimize[T](pgrm: Code[T, {}]): Code[T, {}] = pgrm rewrite {
  case code"val arr = new Array[$ta, $tb]($size); $body" =>
    val a = code"?a: Array[$ta]"
    val b = code"?b: Array[$tb]"
    val body2 = body rewrite {
      case code"arr($i)._1"
        => code"$a($i)"
      case code"arr($i)._2"
        => code"$b($i)"
      case code"arr($i) = ($va, $vb)"
        => code"$a($i) = $va; $b($i) = $vb"
      case code"arr($i)"
        => code"($a($i), $b($i))"
      case code"arr.size"
        => code"$a.size"
    }
    val body3 = body2.subst["arr"](abort())
    code" " "
      val a = new Array[$ta]($size)
      val b = new Array[$tb]($size)
      $body3
    " " "
}
```

Figure 6.1 – A speculative rewrite rule for transforming any array of pairs into two arrays.

```
pgrm = code"if (readInt > 0) { val a = new Array[(Int,String)](3); a(0) = (36,
  \"ok\"); a.size }"
size = code"3"
body = code"(?arr)(0) = (36, \"ok\"); (?arr).size"
body2 = code"(?a)(0) = 36; (?b)(0) = \"ok\"; (?a).size"
body3 = body2
result = code"val a = new Array[Int](3); val b = new Array[String](3); a(0) =
  36; b(0) = \"ok\"; a.size"

optimize(pgrm) → code"if (readInt > 0) { $result }"
```

To understand this example, two key properties of Squid should be noted: 1. holes at the end of a list of statements can be viewed as matching the input greedily; for instance, pattern `code"print(42); $b"` will match a `print` statement and *all* following statements in the current

block;⁸ 2. free variables in patterns will match free variables in the program fragments the patterns are matching.

All statements following the array binding that is matched in the original program are captured into body; they are extruded from their enclosing context, and their references to the bound array are transformed into references to the free variable `arr`. In the inner rewriting, we then match references to `arr` to transform usages of the array as they existed in the original program. Note that this process is hygienic: as it traverses a program, the `rewrite` method *only* extrudes bindings that are matched explicitly in patterns (the other, internal bindings use freshly-generated names), and therefore there is no risk of encountering free variables also named ‘arr’ that referred to different bindings than the one we have matched.

Once the inner rewriting has been applied, the result body2 contains a program fragment where patterns like `arr(i)._1` have been replaced with expressions referring to free variables `a` and `b` — in this case, `(?a)(i)`. All remaining references to `arr` are then searched for using the free variable substitution syntax (which evaluates its second argument lazily), and if any is found the rewriting is aborted.

Notice that in the inner rewriting, we deliberately do not handle patterns of the general form `code" $arr($i) = $v "` where `v` is *not* of the form `(x, y)`. As a consequence, arrays used in such a way are not transformed: not all references to `arr` are removed, and the rewriting is aborted. The rationale is that if the original program stored *already*-tupled values into the array, then perhaps it is not a good idea to do the transformation: it may lead to *more* allocation rather than less.⁹

Finally, note that while the optimization in Figure 6.1 is defined for tuples of two elements only, applying it until it reaches a fixed point will also transform arrays of tuples of more elements, as long as an inductive encoding of tuples is used — for example, `(a, b, c, d)` could be encoded as a composition of nested 2-tuples such as `(a, (b, (c, d)))`. This is actually more or less how tuples are encoded in the new Scala 3 version of the language.

To get a sense of how Squid’s type system and contextual code types help us avoid runtime errors, let us look at some programming mistakes that could be made while writing the transformation:

- omitting to insert `$body3` in the result code fragment: this would give result type `Unit` (i.e., `void` in languages like Java) and the `rewrite` method would complain that the rewriting is not type-preserving;
- using the wrong array in the inner rewriting — for example writing `code" $a($i) "` instead of `code" $b($i) "`: the inner `rewrite` would complain that this case tries to rewrite a term of type `ta` to a term of type `tb` and reject it (fail to compile), as above;

⁸This is because a block like `{a; b; c}` is represented in Squid as `{a; {b; {c}}}`.

⁹In a real-world setting, a more precise analysis with heuristics could determine whether or not to apply the rewriting, depending on the usage patterns found for the array.

- forgetting to define one of `a` or `b` in the final result of the main rewriting: since Squid would notice that the free variable `a` would have never been captured, the result type inferred for the whole rewriting would be¹⁰ `Code[T, {a:Array[_]}]` instead of `Code[T, {}]` (the return type explicitly specified for `optimize`), resulting in a type mismatch;
- using `body2` directly instead of `body3`, without performing the variable substitution: a similar error as above would be raised, as variable `arr` would still be assumed free in `body2`, propagating its context requirement to the result type of the outer rewriting expression;
- forgetting to transform some array operations: this will simply abort the rewriting and leave arrays with unexpected usages untouched, ensuring the safety of our transformation; this case includes when the array “escapes” the current scope, being sent to unknown functions;
- trying to evaluate the `size` program fragment with `size.run`: this results in a compile-time error that reads “cannot prove that <context @ 2:7> ::= {}” (explained in Section 6.4);
- accessing the `size` program fragment from outside of the rewriting (e.g., by leaking it through a mutable variable): since the context requirement of `size` is only defined inside the right-hand side of the rewrite rule, `size`’s type becomes `Code[Int, _]` when viewed from the outside, making the term impossible to close and therefore useless.

6.3 Formalization of the Core Language

This section presents the core of our design for statically-typed analytic quasiquotation with non-lexically-scoped open code, demonstrating its main ideas independently of its Scala implementation.

$\lambda^{\mathbb{B}}$ (pronounced “*lambda-braces*”) is a call-by-value multi-stage λ -calculus with two types of pattern matching on code values: **match**, which simply decomposes a term against a pattern, and **rewrite**, which traverses a term bottom-up, applying some transformation on the way.

6.3.1 Syntax

The syntax of $\lambda^{\mathbb{B}}$ is given in Figure 6.2.

\emptyset is the empty language (with no productions). θ is a *meta*-meta-variable that ranges over meta-variables, and is used to parametrize the productions of q . Thus, q_u denotes the syntax of normal terms, while q_\emptyset refers to terms which do not contain unquotes u , so that $[q_\emptyset]$ is the syntax of *quoted values* (discussed in Section 6.3.3), where the unquotes have been evaluated.

¹⁰ In Scala 2, an underscore in type position stands for an existential: the type `Array[_]` stands for the existentially-quantified type `Array[t] forSome { type t }`.

$\overline{\theta}$ denotes 0 to n repetitions of ' θ ' separated by semicolons. We will use **let** $x : T = t_0$ **in** t_1 as syntactic sugar for $(\lambda x : T. t_1) t_0$. Type ascriptions are used to disambiguate types when necessary, and will be useful to define the semantics of pattern matching in Section 6.3.3.

To simplify the development, we make the usual assumption that α -renaming is used whenever needed to prevent shadowing: a context never contains two distinct bindings $x : T$ and $y : S$ such that $x = y$. This allows us to equate contexts with finite partial functions from variable names to types. For example, we use \emptyset for the empty context $\{\}$ and $\Gamma \cup \Gamma'$ for context extension.

Examples

As a first example of a λ^B program, we give below a simple optimization that transforms an expression of the form `pow x 2` into `x * x`. We assume the existence of constants '`pow`' and '`*`' for integer power and multiplication, respectively:

$$\lambda x : \text{Code Int } \emptyset. x \text{ match } [\text{pow } [y] 2] \Rightarrow [\text{let } z = [y] \text{ in } z * z] \text{ else } x$$

The function above takes a code value x and pattern-matches it against the power-of-2 pattern, binding the program fragment extracted as the base to variable y . If the pattern matches, a program is returned that consists in the binding of the code value represented by y to some variable z , that is then multiplied with itself (this avoids duplicating the computations potentially contained in y). If the pattern does not match, the original code value x is simply returned unchanged.

In λ^B , free variables present in quoted terms do not require a special syntax, so for example `code" (?x: Int) + 1 "` is written just `[x + 1]`.

The `closex` construct makes sure that a term contains no free variable x , otherwise defaulting to the associated `else` branch.

To illustrate the use of open terms and show a speculative rewrite rule, we take inspiration from the rewriting of Figure 6.1, which matches usages of an extruded variable `arr` and replaces them with usages of different free variables `a` and `b`. We assume the language is extended with types '`Array`' and '`PairArray`' (similar to `Array[Int]` and `Array[(Int, Int)]` in Scala), and with constants '`mkPairArray`', '`mkArray`', '`size`', '`get`', and '`first`' with the expected semantics. An incomplete version of the rewriting of Figure 6.1, where we handle only two cases (namely

$t ::= q_u$		Term	
$q_\theta ::=$		(Syntax Template)	
n		Literal	
$ t + t$		Addition	
$ x, y, z$		Variable	
$ t t$		Application	
$ \lambda x : T. t$		Abstraction	
$ [q_u]$		Quote	
$ \text{run } t$		Evaluation	
$ t \text{ match } [t] \Rightarrow t \text{ else } t$	Pattern Matching		
$ t \text{ rewrite } [t] \Rightarrow t$	Term Rewriting		
$ \text{close}_x t \text{ else } t$	Speculative Closure		
$ t : T$	Type Ascription		
$ \theta$	(Syntax Extension)		
$u ::=$			
$[x]$	Variable		
$ [\text{const } x]$	Constant		
		Value	
		$v ::=$	
		n	Literal
		$ \langle \lambda x : T. t, \gamma \rangle$	Closure
		$ [q_\emptyset]$	Quote
		$\gamma ::= \{ \overline{x \mapsto v} \}$	Subs. Context
		Type	
		$T, S ::=$	
		Int	Integer
		$ T \rightarrow T$	Function
		$ \text{Code } T \ C$	Code
		$C, \Gamma ::= \{ \overline{c} \}$	Typ. Context
		$c ::=$	
		$x : T$	Binding
		$ \kappa$	Brand

 Figure 6.2 – Syntax of λ^β .

size and *get-first*), is given below:

```

pgrm rewrite [let arr = mkPairArray [n] in [body]] ⇒
  let a = [a : Array] in
  let body2 = body rewrite [size arr] ⇒ [size [a]]
    rewrite [first (get arr [i])] ⇒ [get [a] [i]] in
  closearr [let a = mkArray [n] in [body2]]
  else [let arr = mkPairArray [n] in [body]]
    
```

The program above proceeds in much the same way as in Figure 6.1. For example, given some $pgrm = [\text{let } x = \text{mkPairArray } 3 \text{ in first (get } x\ 0) + \text{second (get } x\ 1)]$, after the outer pattern matches, we get $body = [\text{first (get } arr\ 0) + \text{second (get } arr\ 1)]$, which is subsequently rewritten into $body_2 = [\text{get } a\ 0 + \text{second (get } arr\ 1)]$, and then close_{arr} is called on a term that still contains references to *arr* (as we are missing the rule to rewrite uses of *second*), aborting the rewriting as expected.

6.3.2 Type System

The typing rules of λ^{B} are presented in Figure 6.3. Typing judgment $\Gamma', \Gamma \vdash t : T$ is read “under inner context Γ and outer context Γ' , t has type T ” (see explanations below). Syntax $\vdash t : T$ is shorthand for $\emptyset \vdash t : T$, and $\Gamma \vdash t : T$ is itself shorthand for $\emptyset, \Gamma \vdash t : T$.

Quoting and unquoting

We use a “double-headed” typing judgment in order to type the term inside a quote as having its own, *inner* context (on the right), while remembering the outer context from outside the quote (on the left).

The inner context is the usual typing context, accounting for free variables. Free variables in a quote may be bound by a lambda abstraction, or may remain free and become part of the quoted term’s context requirements.

The outer context is used to type unquotes, which refer to the context outside of the quotation. Since unquotes can only contain variables $[x]$ and constants $[\text{const } x]$, they cannot be nested, so we only need to carry a single outer context even though λ^{B} is a *multi-stage* language. This syntactic restriction does not incur a loss of generality, as a nested unquote such as $[\dots[f [x_0]]\dots]$ can always be encoded by using an intermediate binding: **let** $x_1 = f [x_0]$ **in** $[\dots[x_1]\dots]$.

Notice how in T-QUOTE, the outer context of t becomes the inner context of $[t]$ while the inner context becomes part of the **Code** type of $[t]$, and how in T-ANTI0, the context parameter of unquoted code has to coincide with the inner context of surrounding code.

Running Code

Rule T-RUN requires the context of program fragment t in **run** t to be empty. This is central to avoiding the occurrence of unbound reference errors at runtime. For example, the term **run** $[x + 1]$ is not typeable, similar to how `code" (?x: Int) + 1 ".run` is rejected by Squid.

Pattern Matching

Rule T-MATCH needs to ensure two important properties:

Unquotes in a pattern capture the local context surrounding them. For example, for some $z : \text{Code Int } \emptyset$ in program z **match** $[\lambda x : \text{Int}. [y] + 1] \Rightarrow y$ **else** $[0]$, the type of *extracted* variable y should be **Code Int** $\{x : \text{Int}\}$; indeed, y can be used to extract terms containing references to x (in particular, when $z = [\lambda x : \text{Int}. x + 1]$ we get $y = [x]$). This is achieved by typing the pattern t_p with outer context Γ' (so that Γ' contains the extracted variables) and then typing the body t_b in the original context Γ extended with Γ' .

Because T-ANTI0 requires the unquoted variable's context parameter to *exactly coincide* with the local context surrounding the unquote, Γ' has to contain variables whose types reflect the exact context from which they were extracted. It is important for T-ANTI0 disallow widening of the unquoted variable's type: though it would make sense in expressions, it would also allow the types of terms extracted from patterns to “forget” about their local context requirements (in the previous example, y could be assigned type **Code Int** \emptyset).¹¹ As a result, an expression like $\lambda x : \mathbf{Code\ Int\ } \emptyset. [\lambda y : \mathbf{Int}. [x]]$ is not typeable, but this is not a practical limitation, since one can use an intermediate binding allowing subsumption (T-SUB) to widen the context of x as needed, as in: $\lambda x : \mathbf{Code\ Int\ } \emptyset. \mathbf{let\ } z : \mathbf{Code\ Int\ } \{y : \mathbf{Int}\} = x \mathbf{\ in\ } [\lambda y : \mathbf{Int}. [z]]$.

Extracted variables propagate the scrutinee's own context requirements. For example, consider $t = [x + 1] \mathbf{match\ } [y + 1] \Rightarrow y \mathbf{else\ } [0]$, which extracts a subterm from a program fragment containing free variable x . Term t should have type **Code Int** $\{x : \mathbf{Int}\}$ since x is free in the result $[x]$. This is achieved by adding the original context C of the scrutinee to the context of each extracted term in Γ' , written Γ'/C and formally defined below. In the example above, $C = \{x : \mathbf{Int}\}$, $\Gamma' = \{y : \mathbf{Code\ Int\ } \emptyset\}$ and so $\Gamma'' = \Gamma'/C = \{y : \mathbf{Code\ Int\ } \{x : \mathbf{Int}\}\}$.

Definition 6.3.1 (Context predication Γ/C).

$$\Gamma/C \stackrel{\text{def}}{=} \{(x : f(T, C)) \mid (x : T) \in \Gamma\} \quad \text{where} \quad f(T, C) = \begin{cases} \mathbf{Code\ } T' (C \cup C') & \text{if } T = \mathbf{Code\ } T' C' \\ T & \text{otherwise} \end{cases}$$

Definition 6.3.2 (Tight typing \vdash^*). We write $\Gamma', \Gamma \vdash^* t : T$ to require that Γ' be a smallest context satisfying the typing judgment $\Gamma', \Gamma \vdash t : T$. More formally,¹²

$$\Gamma', \Gamma \vdash^* t : T \stackrel{\text{def}}{=} \Gamma', \Gamma \vdash t : T \wedge (\nexists \Gamma''. \Gamma'', \Gamma \vdash t : T \wedge |\Gamma''| < |\Gamma'|)$$

In rule T-MATCH, we require pattern code t_p to be typed with the smallest outer context possible (i.e., tight typing). Indeed, by weakening, a pattern such as $t_p = [x] + 1$ could not only be typed with outer context $\{x : \mathbf{Code\ Int\ } \emptyset\}$, but also with, e.g., $\{x : \mathbf{Code\ Int\ } \emptyset; y : \mathbf{Int}\}$. We have to reject the latter, as it would introduce a spurious variable y into the scope of body t_b , whereas no y was actually extracted from t_p . Intuitively, this is because when typing a pattern, the outer context serves as a binder for the extracted variables, whereas when typing an expression, it is used as a normal context, where weakening is in order.

¹¹Another approach could be to change the premise of T-ANTI0 to $\Gamma' \vdash x : \mathbf{Code\ } T \ \Gamma$ and to add a “flag” to the typing judgment that specifies whether we are typing an expression (where T-SUB is allowed to happen), or a pattern (where it is not).

¹²Notation $|\Gamma|$, based on the interpretation of contexts as sets, denotes the number of context members c in Γ .

Rewriting

The rule for rewriting T-RW is similar to T-MATCH. The essential differences are that: T-RW does not require the type of pattern t_p to coincide with that of scrutinee t_s , because the pattern may match any sub-term of the scrutinee; T-RW requires body t_b to be a code value with the same type as the pattern, as any matched subexpression will be replaced by t_b ; and finally, T-RW predicates the local context Γ'' on $C \cup \{\kappa\}$, where κ is some fresh “context brand.” The effect is to introduce κ into the context parameters of all terms extracted from the pattern, which will prevent them from being run and otherwise misused: the only way to eliminate that brand from the context of a term is to use the term as body t_b of the rewriting itself.

As a simple example, consider the program:

$$[\lambda x:\text{Int}. x+1] \text{ rewrite } [\llbracket y \rrbracket + 1] \Rightarrow \text{let } z:\text{Int} = \text{run } y \text{ in } [0]$$

which has to be ill-typed because it tries to run the open term $\llbracket x \rrbracket$ extracted as y . Thankfully, T-RW types y not as **Code Int** \emptyset but as **Code Int** $\{\kappa\}$ where κ is some fresh brand preventing T-RUN from applying. Squid uses a similar mechanism (cf. Section 6.4).

6.3.3 Operational Semantics

As shown in Figure 6.2, values v are either integer literals, lambda abstractions closing over some contexts (closures) or quoted code $\llbracket t \rrbracket$ where t does not contain any immediate unquotes (i.e., unquotes that are not inside a quote), which we write $\llbracket q_\emptyset \rrbracket$.

Value substitution contexts γ map variables to values,¹³ and \models is used to express that a value substitution context *conforms to* or is *consistent with* a typing context.

E-* and Q-* Rules

Figure 6.4 shows the basic big step semantics rules of λ^\emptyset . These rules are of the form $\gamma \vdash t \rightarrow v$, read “under context γ , t evaluates to v .” E-* rules are for current stage code. Q-* rules, which are for next stage code (terms surrounded by one level of quotation), replace immediate unquotes in quoted code by the values to be unquoted; they can be seen as the β -rule(s) for quotes. For example, $\{x \mapsto \llbracket y+1 \rrbracket\} \vdash [\lambda y:\text{Int}. \llbracket x \rrbracket] \rightarrow [\lambda y:\text{Int}. y+1]$.

E-RUN takes code from the next stage and evaluates it as code in the current stage. Therefore, it also evaluates code in the second next stage as code in the next stage, etc. and may trigger more Q-* rules to apply.

¹³We prefer not to use substitution (which is often used in presentations of operational semantics) because our syntax prevents expressions from appearing inside unquotes, which means we could not use a straightforward substitution of expressions for variables. Additionally, contexts interact more intuitively with the semantics of pattern matching, which introduces a set of bindings to be merged with the current context.

Term typing

$\frac{}{\Gamma_0, \Gamma \vdash n : \mathbf{Int}} \text{ T-LIT}$	$\frac{\Gamma_0, \Gamma \vdash t_0 : \mathbf{Int} \quad \Gamma_0, \Gamma \vdash t_1 : \mathbf{Int}}{\Gamma_0, \Gamma \vdash t_0 + t_1 : \mathbf{Int}} \text{ T-PLUS}$	$\frac{\Gamma_0, \Gamma \vdash t : T}{\Gamma_0, \Gamma \vdash (t : T) : T} \text{ T-ASC}$
$\frac{\Gamma_0, \Gamma \cup \{x : T\} \vdash t : S}{\Gamma_0, \Gamma \vdash (\lambda x : T. t) : T \rightarrow S} \text{ T-ABS}$	$\frac{(x : T) \in \Gamma}{\Gamma_0, \Gamma \vdash x : T} \text{ T-VAR}$	$\frac{\Gamma_0, \Gamma \vdash t_f : T \rightarrow S \quad \Gamma_0, \Gamma \vdash t_a : T}{\Gamma_0, \Gamma \vdash t_f t_a : S} \text{ T-APP}$
$\frac{\Gamma, C \vdash t : T}{\Gamma_0, \Gamma \vdash [t] : \mathbf{Code } T C} \text{ T-QUOTE}$	$\frac{(x : \mathbf{Code } T \Gamma) \in \Gamma'}{\Gamma', \Gamma \vdash [x] : T} \text{ T-ANTI0}$	$\frac{(x : \mathbf{Int}) \in \Gamma'}{\Gamma', \Gamma \vdash [\mathbf{const } x] : \mathbf{Int}} \text{ T-ANTI1}$
$\frac{\Gamma_0, \Gamma \vdash t : \mathbf{Code } T (C \cup \{x : S\}) \quad \Gamma_0, \Gamma \vdash t' : \mathbf{Code } T C}{\Gamma_0, \Gamma \vdash \mathbf{close}_x t \text{ else } t' : \mathbf{Code } T C} \text{ T-CLOSE}$	$\frac{\Gamma_0, \Gamma \vdash t : \mathbf{Code } T \emptyset}{\Gamma_0, \Gamma \vdash \mathbf{run } t : T} \text{ T-RUN}$	
$\frac{\Gamma_0, \Gamma \vdash t_s : \mathbf{Code } T C \quad \Gamma'' = \Gamma' / C \quad \Gamma_0, \Gamma \vdash t_e : T \quad \Gamma', C \vdash^* t_p : T \quad \Gamma_0, \Gamma \cup \Gamma'' \vdash t_b : T}{\Gamma_0, \Gamma \vdash t_s \text{ match } [t_p] \Rightarrow t_b \text{ else } t_e : T} \text{ T-MATCH}$		
$\frac{\Gamma_0, \Gamma \vdash t_s : \mathbf{Code } T C \quad \Gamma'' = \Gamma' / (C \cup \{\kappa\}) \quad \kappa \notin C \quad \Gamma' C \vdash^* t_p : T' \quad \Gamma_0, \Gamma \cup \Gamma'' \vdash t_b : \mathbf{Code } T' (C \cup \{\kappa\})}{\Gamma_0, \Gamma \vdash t_s \text{ rewrite } [t_p] \Rightarrow t_b : \mathbf{Code } T C} \text{ T-RW}$	$\frac{\Gamma_0, \Gamma \vdash t : T_0 \quad T_0 <: T_1}{\Gamma_0, \Gamma \vdash t : T_1} \text{ T-SUB}$	

Subtyping

$\frac{T_0 <: T_1 \quad C_0 \subseteq C_1}{\mathbf{Code } T_0 C_0 <: \mathbf{Code } T_1 C_1} \text{ T-CODE}$	$\frac{T_0 <: T_1 \quad T_2 <: T_3}{T_1 \rightarrow T_2 <: T_0 \rightarrow T_3} \text{ T-FUN}$	$\frac{}{T <: T} \text{ T-REFL}$
--	--	----------------------------------

Value typing

$$\frac{\Gamma \models \gamma \quad \Gamma \cup \{x : T\} \vdash t : S}{\Gamma \vdash \langle \lambda x : T. t, \gamma \rangle : T \rightarrow S} \text{ T-CLOS}$$

Context conformance

$$\frac{}{\emptyset \models \emptyset} \quad \frac{\Gamma \models \gamma \quad \vdash v : T}{\Gamma \cup \{x : T\} \models \gamma \cup \{x \mapsto v\}} \quad \frac{\kappa \notin C}{\Gamma' / (C \cup \{\kappa\}) \models \gamma}$$

 Figure 6.3 – Typing and subtyping rules of λ^\exists .

Term evaluation

$\frac{}{\gamma \vdash n \rightarrow n} \text{E-LIT}$	$\frac{\gamma \vdash t_0 \rightarrow n_0 \quad \gamma \vdash t_1 \rightarrow n_1 \quad n_0 + n_1 = n_3}{\gamma \vdash t_0 + t_1 \rightarrow n_3} \text{E-PLUS}$	$\frac{\gamma \vdash t \rightarrow v}{\gamma \vdash t : T \rightarrow v} \text{E-ASC}$
$\frac{}{\gamma \vdash \lambda x : T. t \rightarrow \langle \lambda x : T. t, \gamma \rangle} \text{E-ABS}$	$\frac{(x \mapsto v) \in \gamma}{\gamma \vdash x \rightarrow v} \text{E-VAR}$	$\frac{\gamma \vdash t_a \rightarrow v_a \quad \gamma \vdash t_f \rightarrow \langle \lambda x : T. t, \gamma_f \rangle}{\gamma_f \cup \{x \mapsto v_a\} \vdash t \rightarrow v} \text{E-APP}$
$\frac{\gamma \vdash t \rightarrow [t'] \quad x \notin FV(t')}{\gamma \vdash \mathbf{close}_x t \mathbf{else} t_e \rightarrow [t']} \text{E-CLOSED}$	$\frac{\gamma \vdash t \rightarrow [t'] \quad x \in FV(t')}{\gamma \vdash t_e \rightarrow v} \text{E-OPEN}$	$\frac{\gamma \vdash t \rightarrow [t']}{\gamma \vdash t' \rightarrow v} \text{E-RUN}$
$\frac{}{\gamma \vdash t_s \mathbf{match} [t_p] \Rightarrow t_b \mathbf{else} t_e \rightarrow v} \text{E-MATCH}$	$\frac{\gamma \vdash t_s \rightarrow [t'_s] \quad t'_s \gg t_p = \gamma_b}{\gamma \cup \gamma_b \vdash t_b \rightarrow v} \text{E-NOMATCH}$	$\frac{\gamma \vdash t_s \rightarrow [t'_s] \quad (t'_s, t_p) \notin \text{dom}(\gg)}{\gamma \vdash t_e \rightarrow v} \text{E-NOMATCH}$

Quote evaluation

$\frac{}{\gamma \vdash [n] \rightarrow [n]} \text{Q-LIT}$	$\frac{\gamma \vdash [t_0] \rightarrow [t'_0] \quad \gamma \vdash [t_1] \rightarrow [t'_1]}{\gamma \vdash [t_0 + t_1] \rightarrow [t'_0 + t'_1]} \text{Q-PLUS}$	$\frac{\gamma \vdash [t] \rightarrow [t']}{\gamma \vdash [t : T] \rightarrow [t' : T]} \text{Q-ASC}$
$\frac{\gamma \vdash [t] \rightarrow [t']}{\gamma \vdash [\lambda x : T. t] \rightarrow [\lambda x : T. t']} \text{Q-ABS}$	$\frac{}{\gamma \vdash [x] \rightarrow [x]} \text{Q-VAR}$	$\frac{\gamma \vdash [t_f] \rightarrow [t'_f] \quad \gamma \vdash [t_a] \rightarrow [t'_a]}{\gamma \vdash [t_f t_a] \rightarrow [t'_f t'_a]} \text{Q-APP}$
$\frac{}{\gamma \vdash [[t]] \rightarrow [[t]]} \text{Q-QUOTE}$	$\frac{(x \mapsto n) \in \gamma}{\gamma \vdash [\mathbf{const} x] \rightarrow [n]} \text{Q-ANTI0}$	$\frac{(x \mapsto [t]) \in \gamma}{\gamma \vdash [[x]] \rightarrow [t]} \text{Q-ANTI1}$
$\frac{\gamma \vdash [t] \rightarrow [t'] \quad \gamma \vdash [t_e] \rightarrow [t'_e]}{\gamma \vdash [\mathbf{close}_x t \mathbf{else} t_e] \rightarrow [\mathbf{close}_x t' \mathbf{else} t'_e]} \text{Q-CLOSE}$	$\frac{}{\gamma \vdash [\mathbf{run} t] \rightarrow [\mathbf{run} t']} \text{Q-RUN}$	
$\frac{\gamma \vdash [t_s] \rightarrow [t'_s] \quad \gamma \vdash [t_b] \rightarrow [t'_b] \quad \gamma \vdash [t_e] \rightarrow [t'_e]}{\gamma \vdash [t_s \mathbf{match} [t_p] \Rightarrow t_b \mathbf{else} t_e] \rightarrow [t'_s \mathbf{match} [t_p] \Rightarrow t'_b \mathbf{else} t'_e]} \text{Q-MATCH}$		
$\frac{\gamma \vdash [t] \rightarrow [t'] \quad \gamma \vdash [t_b] \rightarrow [t'_b]}{\gamma \vdash [t \mathbf{rewrite} [t_p] \Rightarrow t_b] \rightarrow [t' \mathbf{rewrite} [t_p] \Rightarrow t'_b]} \text{Q-RW}$		

 Figure 6.4 – Main rules of the big step operational semantics of λ^{\dagger} .

Chapter 6. Improved Safety and Expressivity for Analytic Metaprogramming

Rules E-MATCH and E-NOMATCH make use of partial function \gg to match a term against a given pattern, producing a value substitution context that contains the results of the matching. For example, we have:

$$[(x : \mathbf{Int} \rightarrow \mathbf{Int}) (123 : \mathbf{Int})] \gg^{\mathbf{Int}} [(\lambda y : \mathbf{Int} \rightarrow \mathbf{Int}) (\mathbf{const } z : \mathbf{Int})] = \{y \mapsto [x : \mathbf{Int}]; z \mapsto 123\}$$

The definitions of \gg and \gg^T are explained later in this section.

Rule E-CLOSED searches for free occurrences of x in its argument term (where FV is defined as usual for multi-stage languages, for example see [Rhiger, 2012a]); it evaluates to the term unchanged if there are none, and otherwise evaluates to the **else** branch.

For lack of space, we do not list all the rules for **rewrite**, which simply go through all the subterms of a term and transform it by reusing the semantics of pattern matching. We only see three examples below:

<p style="text-align: center;">RW-LIT</p> $\frac{\gamma \vdash t \rightarrow [n] \quad \gamma \vdash [n] \mathbf{match} [t_p] \Rightarrow t_b \mathbf{else} [n] \rightarrow v}{\gamma \vdash t \mathbf{rewrite} [t_p] \Rightarrow t_b \rightarrow v}$	<p style="text-align: center;">RW-PLUS</p> $\frac{\begin{array}{l} \gamma \vdash t \rightarrow [t_0 + t_1] \\ \gamma \vdash [t_0] \mathbf{rewrite} [t_p] \Rightarrow t_b \rightarrow [t'_0] \\ \gamma \vdash [t_1] \mathbf{rewrite} [t_p] \Rightarrow t_b \rightarrow [t'_1] \\ \gamma \vdash [t'_0 + t'_1] \mathbf{match} [t_p] \Rightarrow t_b \mathbf{else} [t'_0 + t'_1] \rightarrow v \end{array}}{\gamma \vdash t \mathbf{rewrite} [t_p] \Rightarrow t_b \rightarrow v}$
---	--

Rule RW-LIT applies pattern matching on a constant literal n (as this term has no sub-expressions); if the pattern does not match, the rule returns the term unchanged. Rule RW-PLUS first applies **rewrite** recursively inside both sides of an addition, and then applies pattern matching to transform the top-level expression made of the results of these two recursive calls.

RW-ABS

$$\frac{\begin{array}{l} \gamma \vdash t \rightarrow [\lambda x : T. t_0] \quad t'_0 = [x \mapsto y] t_0 \quad y \text{ fresh} \\ \gamma \vdash [t'_0] \mathbf{rewrite} [t_p] \Rightarrow t_b \rightarrow [t''_0] \quad \gamma \vdash [\lambda y : T. t''_0] \mathbf{match} [t_p] \Rightarrow t_b \mathbf{else} [n] \rightarrow v \end{array}}{\gamma \vdash t \mathbf{rewrite} [t_p] \Rightarrow t_b \rightarrow v}$$

More interestingly, RW-ABS makes sure to change the name of the variable bound by the lambda term it rewrites, before recursing in its body. This way, the names of bound variables inside rewritten programs can never clash with other names.

Intensional Type Analysis

Similar to Squid (cf. Section 3.9.4), λ^\S performs run-time subtyping checks to guide pattern matching. For example, pattern $[(\lambda x : \mathbf{Int} \rightarrow \mathbf{Int}) [y]]$, where x and y are typed respectively as **Code** $(\mathbf{Int} \rightarrow \mathbf{Int})$ C and **Code** \mathbf{Int} C , should not match a program fragment such as $[(\lambda x_0 : \mathbf{Int} \rightarrow \mathbf{Int}. x_0) (\lambda x_1 : \mathbf{Int}. x_1)]$, because the extracted terms would not have the correct types expected by the pattern.

$(t : T) \gg (t' : S)$	$= t \gg^T t' \text{ if } T <: S$	(X-ASC)
$x \gg^T x$	$= \emptyset$	(X-VAR)
$t \gg^T [x]$	$= \{x \mapsto [t : T]\}$	(X-ANTI0)
$n \gg^T n$	$= \emptyset$	(X-LIT)
$n \gg^T [\mathbf{const} \ x]$	$= \{x \mapsto n\}$	(X-ANTI1)
$[t] \gg^T [t]$	$= \emptyset$	(X-QUOTE)
$(\lambda x : S. t) \gg^T (\lambda y : S'. t')$	$= [z \mapsto y]([x \mapsto z] t \gg [y \mapsto z] t') \quad z \text{ fresh}$	(X-ABS)
$(\mathbf{run} \ t) \gg^T (\mathbf{run} \ t')$	$= t \gg^T t'$	(X-RUN)
$(t_0 + t_1) \gg^T (t'_0 + t'_1)$	$= (t_0 \gg^T t'_0) \uplus (t_1 \gg^T t'_1)$	(X-PLUS)
$(t_0 \ t_1) \gg^T (t'_0 \ t'_1)$	$= (t_0 \gg^T t'_0) \uplus (t_1 \gg^T t'_1)$	(X-APP)
$(\mathbf{close}_x \ t_0 \ \mathbf{else} \ t_1) \gg^T$		
$(\mathbf{close}_x \ t'_0 \ \mathbf{else} \ t'_1)$	$= (t_0 \gg^T t'_0) \uplus (t_1 \gg^T t'_1)$	(X-CLOSE)
$(t_s \ \mathbf{match} \ [t_p] \Rightarrow t_b \ \mathbf{else} \ t_e) \gg^T$		
$(t'_s \ \mathbf{match} \ [t_p] \Rightarrow t'_b \ \mathbf{else} \ t'_e)$	$= (t_s \gg^T t'_s) \uplus (t_b \gg^T t'_b) \uplus (t_e \gg^T t'_e)$	(X-MATCH)
$(t_s \ \mathbf{rewrite} \ [t_p] \Rightarrow t_b) \gg^T$		
$(t'_s \ \mathbf{rewrite} \ [t_p] \Rightarrow t'_b)$	$= (t_s \gg^T t'_s) \uplus (t_b \gg^T t'_b)$	(X-RW)

 Figure 6.5 – Extraction rules for pattern matching in λ^\exists .

In order to enable those runtime checks, we actually perform evaluation not directly on a source program t , but on its translation $\llbracket t \rrbracket_{\Gamma}^{\Gamma'}$ into an explicitly-typed variant of λ^\exists — a form where every subterm is annotated with its type as assigned by the typing rules, given inner context Γ and outer context Γ' . For example, $\llbracket x + 1 \rrbracket_{\{x : \mathbf{Int}\}}^{\emptyset} = (x : \mathbf{Int}) + (1 : \mathbf{Int}) : \mathbf{Int}$.

The $\llbracket t \rrbracket_{\Gamma}^{\Gamma'}$ function traverses t in lockstep with the typing rules and adds type annotations to each subterm. Below, we show a few cases — the other cases are straightforward:

A-IDEM $\frac{}{\llbracket t : T_0 \rrbracket_{\Gamma}^{\Gamma'} = (t : T_0)}$	A-GROUND $\frac{\Gamma', \Gamma \vdash t : T \quad t \in \{n; x; [\mathbf{const} \ x]; [x]\}}{\llbracket t \rrbracket_{\Gamma}^{\Gamma'} = (t : T)}$
A-APP $\frac{\Gamma', \Gamma \vdash t_0 \ t_1 : T}{\llbracket t_0 \ t_1 \rrbracket_{\Gamma}^{\Gamma'} = (\llbracket t_0 \rrbracket_{\Gamma}^{\Gamma'} \ \llbracket t_1 \rrbracket_{\Gamma}^{\Gamma'} : T)}$	A-QUOTE $\frac{\Gamma', \Gamma \vdash [t] : \mathbf{Code} \ T \ C}{\llbracket [t] \rrbracket_{\Gamma}^{\Gamma'} = \llbracket [t'] \rrbracket_{\Gamma}^{\Gamma'} : \mathbf{Code} \ T \ C}$

etc.

Remark that $\llbracket \cdot \rrbracket_{\Gamma, C}^{\Gamma'}$ is idempotent, thanks to A-IDEM, so there is always a *single* ascription on each subterm.

Extraction

Figure 6.5 shows the definitions of partial function \gg and its helper \gg^T . We write $\gamma_0 \uplus \gamma_1$ for the disjoint union of value substitution contexts γ_0 and γ_1 , which is only defined if their domains are disjoint, that is to say, $\text{dom}(\gamma_0) \cap \text{dom}(\gamma_1) = \emptyset \implies \gamma_0 \uplus \gamma_1 \stackrel{\text{def}}{=} \gamma_0 \cup \gamma_1$.

Case X-VAR matches two variables with the same name and compatible types, producing an empty result (as nothing is extracted from this match). For example $[x : \text{Code Int } \emptyset]$ matches pattern $[x : \text{Code Int } \{y : \text{Int}\}]$ because we have $\text{Code Int } \emptyset <: \text{Code Int } \{y : \text{Int}\}$. On the other hand, case X-ANTI0 matches anything with a compatible type and *extracts* it as a code value, which corresponds to the semantics of unquotes in pattern position.

Particularly interesting is the X-ABS case, which matches two lambda bodies by renaming both variable x bound in the scrutinee and variable y bound in the pattern to some fresh variable z , and then renaming all occurrences of z to y in the result. This way, any code extracted by this rule will refer to bound variable x of the original term as y , the name used in the pattern. Remember that we assume sufficient α -renaming to avoid name collisions, which includes the assumption that y is not already *free* in t — if it was, we would get conflicting contexts while typing the pattern, between the y bound in the pattern and the y coming from the scrutinee of the pattern match or rewriting.

X-ABS effectively makes the names of bound variables irrelevant to the operational semantics of λ^\exists . Interestingly, this gives us a way to compare open terms for α equivalence — in Squid, it is implemented as reciprocal matching $t_0 \equiv_\alpha t_1 \stackrel{\text{def}}{=} (t_0 \gg t_1) = (t_1 \gg t_0) = \emptyset$.

6.3.4 Soundness of λ^\exists

We now look into proving the safety of λ^\exists .

Top-Level Evaluation

We write $t \Downarrow v$ the annotation and evaluation of program t down to value v , an abbreviation of $\emptyset \vdash \llbracket t \rrbracket_\emptyset^\emptyset \rightarrow v$. Note that in the proofs below, we refer to terms t with no assumptions on whether they are in an annotated form or not, because that is not a requirement for the soundness of λ^\exists . Failing to annotate a program before evaluating it will *not* result in evaluation getting stuck, however it *may* result in a different evaluation result due to some patterns not matching, as partial function \gg is not defined on terms lacking explicit type annotations.

Canonical Forms

Since the type system admits a subtyping rule (T-SUB) and a reflexive subtyping relation, inverting the typing judgment always yields multiple possibilities, including the use of T-SUB. This leads to some bureaucracy in the proofs, forcing us to take care of the subtyping case in

addition to the main case.

To help with that issue, we introduce an inversion lemma for the subtyping relation:

Lemma 6.3.3 (Subtyping inversion). *If $S <: T$, then S is a pointwise-subtype of T , define as:*

- *S has the same type constructor as T*
- *Then, depending on T :*
 - *if $T = \mathbf{Int}$, then $S = \mathbf{Int}$*
 - *if $T = T_1 \rightarrow T_2$, then $S = S_1 \rightarrow S_2$ with $T_1 <: S_1$ and $S_2 <: T_2$*
 - *if $T = \mathbf{Code} \ T' \ C_T$, then $S = \mathbf{Code} \ S' \ C_S$ with $S' <: T'$ and $C_S \subseteq C_T$*

Proof. By induction on derivations of $S <: T$. □

Notice that because S and T are so tightly coupled, the "upward" version of lemma 6.3.3 is also admissible (where we do a case analysis on S to infer T 's shape).

Thanks to this lemma, we know that T-SUB preserves the type constructor and can only replace its arguments by subtypes of theirs (or supertypes in contravariant positions). In the following, when it is clear that a property is preserved by subtyping thanks to this lemma, we may use the phrase "modulo subtyping" as a shorthand.

Remark 1 (Inversion Modulo Subtyping). *Lemma 6.3.3 has an important consequence. First, notice that for any term shape t , only one typing rule \mathcal{R} applies aside from T-SUB — essentially, the system is syntax-directed modulo subtyping. Thus, we know precisely the structure of any type derivation for terms of that shape: it ends with \mathcal{R} followed by an arbitrary number of instances of T-SUB.*

Now, applying the lemma to that observation means that all instances of T-SUB in that derivation yield pointwise-subtypes, which is a reflexive and transitive relation. Therefore, inverting the typing assumption yields the use of \mathcal{R} , just slightly weakened — the type of t is replaced by an arbitrary pointwise-subtype (both in the premises and the conclusion).

Lemma 6.3.4 (Preservation for annotation). *If $\Gamma', \Gamma \vdash t : T$, then $\Gamma', \Gamma \vdash \llbracket t \rrbracket_{\Gamma}^{\Gamma'} : T$.*

Proof. By induction on derivations of $\Gamma', \Gamma \vdash t : T$ and definition of $\llbracket t \rrbracket_{\Gamma}^{\Gamma'}$. □

Lemma 6.3.5 (Canonical Forms). *If $\vdash v : T$, then:*

- *if $T = \mathbf{Int}$, then $v = n$ for some n ;*
- *if $T = T_1 \rightarrow T_2$, then $v = \langle \lambda x : T_1. t, \gamma \rangle$ for some x, t , and γ ;*
- *if $T = \mathbf{Code} \ T_1 \ C$, then $v = [\![t]\!]$, for some t such that $C \vdash t : T_1$.*

Proof. By induction on the typing rules and the syntax of values, modulo subtyping. □

We will also need the following lemma about the type of a quote.

Lemma 6.3.6 (Inversion for quotes). *If $\Gamma \vdash [\![t]\!] : T$, then there exists T_0, T_1, C_0, C_1 such that $T = \mathbf{Code} \ T_1 \ C_1$ and $\Gamma, C_0 \vdash t : T_0$, with $T_0 <: T_1$ and $C_0 \subseteq C_1$.*

Proof. By induction on the typing derivation. There are only 2 cases that apply: T-SUB, which is immediate by 6.3.3 and by transitivity of both $<:$ and \subseteq ; and the base case, T-QUOTE, which allows to conclude thanks to the reflexivity of those relations. \square

For the proof of preservation, we first need the following lemma.

Lemma 6.3.7 (Evaluation to quotes yields values). *For any value substitution context γ and term t , if $\gamma \vdash t \rightarrow [\![t']]\!$, then $[\![t']]\! \in v$. In other words, a term never evaluates to a quote containing immediate unquotes.*

Proof. By induction on the reduction. E-ASC and Q-ASC are immediate by induction on their unique premise, E-CLOSED on its first. All the other E-* rules are immediate, since none produces an unspecified quoted term; thus, all the rules which can apply (producing a value v) are obviously correct. The only interesting Q-* rule is Q-ANTI0, which is solved by the observation that γ maps identifier to values, applied to the first premise. Q-ANTI1 is trivial since the resulting quote is a value. All the other Q-* rules are solved directly because they are essentially congruence rules — under the assumption that no subterm contains an immediate unquote after reduction, then the term itself can't contain one either (after reduction, again). \square

Lemma 6.3.8 (Extraction weak conformance). *Extraction in conforming contexts yields weakly-conforming contexts: If we have*

- $\Gamma \models \gamma$;
- $\Gamma \vdash t_s : \mathbf{Code} \ T \ C$;
- $\Gamma', C \vdash^* t_p : T$;
- $\Gamma'' = \Gamma' / C$;
- $\gamma_b = t_s \gg t_p$;

then $\Gamma \cup \Gamma'' \models_d \gamma \cup \gamma_b$, where we write weak conformance $A \models_d B$ as a shorthand for:

$$A \models_d B \stackrel{\text{def}}{=} \{x \mapsto v \mid (x \mapsto v) \in A \wedge x \in \text{dom}(B)\} \models B$$

Proof. By induction on the extraction rules. It is easy to see that each extraction rule matches subterms of the same types in conforming contexts, which allows applying the induction hypothesis. Notice that \models_d is weaker than \models (as used in T-MATCH) — this is necessary to allow the base cases of the induction, which may individually extract *fewer* bindings than are present in the full typing context. The interesting cases of the induction are X-ANTI0 and X-ANTI1, which are the only rule to add bindings to the extraction result (easily shown to

weakly conform). Rule X-ABS recurses on lambda bodies with a renamed parameter variable, and it is an easy result that renaming with a fresh variable preserves typing. \square

Lemma 6.3.9 (Extraction completeness). *Extraction provides bindings for all binding in the typing context of the tightly-typed pattern: Under the same premises as in lemma 6.3.8, we have:*

$$\text{dom}(\Gamma'') \subseteq \text{dom}(\gamma_b)$$

Proof. By induction on derivations of typing and tight typing. In each case, we can tightly type each sub-pattern and then weaken the corresponding Γ' just enough to tightly type the whole pattern (preservation of typing under weakening is straightforward to prove). \square

Together, lemmas 6.3.8 and 6.3.9 show that extraction in the context of the pattern matching typing and evaluation rules yields fully-conforming contexts $\Gamma \cup \Gamma'' \models \gamma \cup \gamma_b$.

Lemma 6.3.9 is interesting, because it demonstrates why we need a *tight* typing of t_p in T-MATCH. Indeed, if we allowed Γ' to be wider than strictly necessary, then we could end up with bindings that are bound when typing the body of the pattern matching, but never actually extracted from the pattern.

Lemma 6.3.10 (Preservation — general). *Evaluation in conforming contexts preserves typing: If $\gamma \vdash t \rightarrow v$ then for all Γ, Γ', T such that $\Gamma \models \gamma$ and $\Gamma', \Gamma \vdash t : T$, we have $\Gamma \vdash v : T$.*

Proof. By induction on the evaluation derivation. In the following, we replace t by the notations used in the conclusions of the typing & evaluation rules.

Case E-VAR Since x is typable in Γ , then $\Gamma(x) = T$. We conclude by conformance of γ to Γ .

Case E-LIT, Q-LIT, Q-VAR, Q-QUOTE These cases are immediate since v evaluates to itself.

Case E-IGNORE By remark 1, inverting the judgment $\Gamma \vdash (t : T) : T \dashv \Gamma'$ yields $\Gamma \vdash t : S \dashv \Gamma'$, with $S <: T$. By induction hypothesis on that judgment, v has type S as well — and T by T-SUB.

Case E-ABS By inversion modulo subtyping, we get that $T = T_1 \rightarrow T_2$ and that there exists S_1 and S_2 such that $T_1 <: S_1$, $S_2 <: T_2$ and $\Gamma \vdash \lambda x : S_1. t : S_1 \rightarrow S_2$. The resulting closure v is typed via T-CLOS and T-SUB; the second premise of T-CLOS is exactly the same as the one of T-ABS, and the first premise is provided by the assumption that $\Gamma \models \gamma$.

Case E-APP By remark 1, we inverse the typing judgment and obtain the judgments $\Gamma \vdash t_f : T' \rightarrow S'$ and $\Gamma \vdash t_a : T''$, with $S' <: S$ and $T'' <: T <: T'$. By induction on the first 2 premises, the closure has the same type than t_f , namely $T' \rightarrow S'$, and v_a has the same type than t_a , T' . Notice that $\Gamma \cup \{x : T\} \models \gamma \cup \{x \mapsto v_a\}$. The context exactly coincides with the one of rule T-CLOS for the closure (by inversion and remark 1 again). The conclusion follows by induction on the last premise.

Case E-PLUS By inversion of the typing judgment, we get that both t_0 and t_1 have type **Int**. By induction and the canonical forms lemma, they reduce to two constants n_0 and n_1 .

Case E-RUN By the inversion lemma for quotes, the type of t is **Code** $T' C'$ for some T' and C' . By induction on the first premise, $\llbracket t' \rrbracket$ (which is a value) has the same type as $\llbracket t \rrbracket$, namely **Code** $T' C'$. This in turn gives us that t' has type S , with $S <: T'$. Hence, by induction on the second premise, v has type S ; thus also type T' by T-SUB — and T' is also the type of the term **run** t .

Case E-MATCH By inversion modulo subtyping, the base case must be T-MATCH with some return type S , where $S <: T$. This gives us the right typing judgment on t_b to use with the corresponding induction hypothesis (on the evaluation of t_b), thanks to lemmas 6.3.8 and 6.3.9 which show that $\Gamma \cup \Gamma'' \models \gamma \cup \gamma_b$. Thus, v has type S , and also T by T-SUB.

Case E-NO MATCH This case is similar to the last one, but simpler (no context extension).

Cases E-Rw-* All these rules are handled in a similar fashion. First, by inversion modulo subtyping, we get that the base case must be T-RWR. We also apply the induction hypothesis to the first premise, the one asserting the reduction of the scrutinee — invoking lemma 6.3.7 if necessary. This allows, after inverting (modulo subtyping) the typing hypothesis we just derived, to apply induction on any sub-rewrite premise. This ensures that the subterms of the scrutinee (in the last premise) have the correct type. One concludes by induction on the last premise.

Case E-CLOSED By inversion modulo subtyping, one gets $\Gamma \vdash t : \text{Code } T' C' \dashv \Gamma'$, with $T' <: T$ and $C' \subseteq C \cup \{x : S\}$. By induction on the first premise, one get that $\llbracket t' \rrbracket$ has the same type. If $x : S \in C'$, it is easy to see that $\llbracket t' \rrbracket$ can also be given the subtype **Code** $T' (C' \setminus \{x : S\})$, by the second premise of E-CLOSED. We conclude by applying T-SUB if necessary.

Case E-OPEN By inversion modulo subtyping, one gets $\Gamma \vdash t : \text{Code } T' C' \dashv \Gamma'$, with $T' <: T$ and $C' \subseteq C$. We conclude by induction hypothesis on the last premise, using T-SUB if necessary.

Case Q-ANTI0 Again by remark 1, inverting the typing judgment on $\llbracket [x] \rrbracket$ yields that it has type **Code** $T' C'$, and that $C' \vdash [x] : T' \dashv \Gamma$ with $T' <: T$. Inverting that premise again gives us $x : \text{Code } T'' C' \in \Gamma$ with $T'' <: T'$. Since $x \mapsto \llbracket t \rrbracket \in \gamma$ (premise of Q-ANTI0) and $\Gamma \models \gamma$, we get that $\llbracket t \rrbracket$ is a value of type **Code** $T'' C'$ in Γ . We conclude by applying T-SUB if necessary.

Case Q-ANTI1 By inversion modulo subtyping and since **Int** has itself as only subtype/supertype, we have $\Gamma \vdash \llbracket [\text{const } x] \rrbracket : \text{Code Int } C$, as well as $C \vdash [\text{const } x] : \text{Int} \dashv \Gamma$ (T-QUOTE), and $x : \text{Int} \in \Gamma$ (T-ANTI1). We conclude by recalling that $\Gamma \models \gamma$.

All the remaining Q-* cases, which all apply to quoted terms, are handled the same way. Each of these cases has premises of the form $\llbracket t \rrbracket \rightarrow \llbracket t' \rrbracket$. Thanks to 6.3.7, we show that such

$[t']$ terms are always values. Thus, we get an induction hypothesis for all such premises (since, by inversion modulo subtyping, the base case for all premises is always T-QUOTE), and we conclude by mirroring the input type derivation for the reduced term, applying T-SUB whenever necessary. \square

We get preservation as an immediate corollary:

Theorem 6.3.11 (Type Preservation). *If $\vdash t : T$ and $t \Downarrow v$, then $\vdash v : T$.*

Proof. By Lemmas 6.3.4 and 6.3.10. Notice that by definition $\emptyset \models \emptyset$. \square

In big step semantics, to distinguish between terms diverging and terms getting stuck, it is customary to extend the syntax with an *error* value **err** (syntax $v_e ::= v \mid \mathbf{err}$) and add rules to, on the one hand, generate errors when no original rule applies, and on the other to propagate errors. Then, *progress* is the property that if a well-typed program evaluates to a value, that value is not an error. The error-related rules for λ^0 are standard, unsurprising, and therefore omitted from this presentation.

For the proof of progress, we first need a version of it that only applies to quoted terms, and assert that they all reduce to quoted terms.

Lemma 6.3.12 (Quote Progress). *For any contexts Γ and Γ' , value substitution context γ such that $\Gamma \models \gamma$, and every term t such that $\Gamma', \Gamma \vdash t : T$, there exists t' such that $\gamma \vdash [t] \rightarrow [t']$.*

Proof. By induction on $\Gamma \vdash t : T$. T-ANTI0 and T-ANTI1 both work thanks to their corresponding Q-* rules and the assumption that $\Gamma \models \gamma$. The base cases T-VAR, T-LIT and T-QUOTE are trivial by the associated Q-* rules. All the other cases are equally easy, since they don't affect the outer context, and corresponding Q-* rules act as congruences. Notice that this remark also apply to T-MATCH and T-RWR — the only premise where they modify the outer contexts are for the branches, but these are also left untouched by the associated Q-* rules. \square

Finally, we will also rely implicitly on the fact that the evaluation relation is deterministic.

Lemma 6.3.13 (Progress — general). *Assume $\Gamma \models \gamma$. For any fully annotated term t , if $\Gamma \vdash t : T$ and $\gamma \vdash t \rightarrow v_e$, then $v_e \neq \mathbf{err}$.*

Proof. By induction on the typing and conformance derivations. To be more precise, we use a strong induction on the size of the typing and context conformance derivations. Most cases can be solved simply by a structural induction, and we handle them in this style; but one case (T-RUN) requires a slightly more general induction principle.

Case T-VAR Immediate by conformance of γ to Γ .

Case T-LIT Immediate since t is a non-quote value (and these never step).

Case T-RUN From the unique premise, $\Gamma \vdash t : \mathbf{Code} \ T \ \emptyset$. By induction hypothesis, we know that $\gamma \vdash t \rightarrow v$ for some v , and by preservation, $\Gamma \vdash v : \mathbf{Code} \ T \ \emptyset$. By canonical form, $v = [t']$ for some t' such that, by quote inversion modulo subtyping, $\vdash t' : S$ for some $S <: T$. We now need to apply the induction hypothesis on $\vdash t' : S$. To do that, we leverage the fact that a typing/conformance derivation for this judgement is necessarily of size smaller than or equal to the derivation for $\Gamma \vdash [t'] : \mathbf{Code} \ T \ \emptyset$ (immediate by inversion of typing judgements), which is itself of size smaller than or equal to the derivation for the premise $\Gamma \vdash t : \mathbf{Code} \ T \ \emptyset$ (this can be proven in a separate lemma, stating that evaluation reduces the size of typing/conformance derivations). Hence, by induction, we get that for all $v_e^{t'}$ such that $\gamma \vdash s \rightarrow v_e^{t'}$, $v_e^{t'}$ is not an error; and $v_e = v_e^{t'}$.

Case T-ABS t evaluates to the corresponding closure, by rule E-ABS (and closures are values).

Case T-APP By induction on the premises and preservation, we get 2 values of the corresponding types. By canonical forms on the value obtained for t_f , it is a closure with the appropriate argument type (or a supertype thereof). By induction on its typing derivation, the body reduces without error when the argument is added to the context. We conclude via E-APP.

Case T-ASC Immediate from the induction hypothesis on the unique premise.

Case T-QUOTE By combining lemmas 6.3.7 and 6.3.12.

Cases T-ANTI0 and T-ANTI1 Impossible since the right context is empty.

Case T-PLUS By the induction, inversion modulo subtyping, canonical forms and E-PLUS.

Case T-MATCH By induction on the first premise, if the scrutinee evaluates to a value, it is not an error. By preservation and canonical forms, this value is a quoted term. Depending on the result of the extraction check, we keep evaluating either t_b or t_e . By induction on the two corresponding premises of T-MATCH, either cases evaluate safely, and we can apply E-MATCH. To use the induction hypothesis for t_b we use lemmas 6.3.8 and 6.3.9, which show the conformance of the corresponding contexts.

Case T-RWR Like in T-MATCH, the scrutinee evaluates safely (if ever). The reduction is split across multiple rules, but the reasoning is essentially the same as in the pattern matching case. One may have to perform more inversion on the typing hypothesis (to match the structure of the rewriting being performed — this is proved by induction on the rewriting rules), and conclude from the associated induction hypotheses.

Case T-SUB Follows immediately from the induction hypothesis.

Case T-CLOSE By induction (first premise), preservation and canonical forms, if $t \rightarrow v$ then v has shape $[t']$. Then, check whether $x \in FV(t')$. If not, an easy result shows that $[t']$

also has type **Code** $T\ C$ (removing the uses of T-SUB that add x). We conclude by E-CLOSED. If, on the other hand, x is in $FV(t')$, we conclude by induction (second premise of T-CLOSE) and E-OPEN.

□

Theorem 6.3.14 (Progress). *If $\vdash t : T$ and $t \Downarrow v_e$, then $v_e \neq \text{err}$.*

Proof. By Lemma 6.3.13. Notice that by definition $\emptyset \models \emptyset$.

□

6.4 Implementation in Scala

We now briefly describe how the mechanisms of λ^{B} are implemented via Squid inside the Scala programming language.

Contexts as contravariant structural types. The type of quoted terms `Code[T, C]` is defined in Scala as `type Code[+Typ, -Ctx]`. From the + and - prefixes, we can see that this type is covariant in its `Typ` parameter, and contravariant in `Ctx`. Scala applies the traditional rules for structural subtyping so that, for example, for all types X and Y where $X \supset Y$, we have $\{\} \supset \{a : X\} \supset \{a : Y\}$. Therefore, by Scala's subtyping rules, for all T we also have:

$$\text{Code}[Y, \{\}] <: \text{Code}[Y, \{a : X\}] <: \text{Code}[Y, \{a : Y\}] <: \text{Code}[X, \{a : Y\}]$$

In other words, a term that requires some context C can be used in place of a term that requires some more specific context $D <: C$. In particular, a closed term, which requires no context (written $\{\}$), can be used in place of a term that requires any context. This subsumption principle is also called *weakening* [Rhiger, 2012a] or *type widening*, and is important for the flexibility of the quasiquotes API. It is directly reflected in λ^{B} by typing rule T-SUB presented in Section 6.3.2.

Context polymorphism. A consequence of this encoding of contexts as Scala types means that in Squid we can abstract over contexts the same way we abstract over other types. This capability is known as *context* or *scope polymorphism* (also called *support polymorphism* [Nanevski, 2002]). However, we will see that a naive interpretation of context polymorphism in λ^{B} and Squid leads to unsoundness (and lack of binder hygiene). The goal of Chapter 7 is to extend λ^{B} and its Squid implementation to soundly support context polymorphism.

Type intersection and structural refinement. Scala has a concept of intersection types, where $A \ \& \ B$ represents the intersection of types A and B . This means we can express *refinements* on abstract contexts by intersecting an abstract context parameter C and a structural type such as $\{x : T\}$, as in $C \ \& \ \{x : T\}$, also simply written $C\{x : T\}$.

Term rewriting. Squid’s `rewrite` macro, which allows the recursive transformation of all subterms of a program, has to make sure that intermediate extracted subterms are only used in the context from where they were extracted, otherwise it could result in unsafe scope extrusion. This is achieved by making the context matched by each case of the rewrite rule a refinement on some abstract context type that is only usable within the pattern matching branch. For example, in the program below, the type of extracted variables `n` and `m` is `Code[Int, <context @ 2:15>{y:Int}]`. The type `<context @ 2:15>`, where `2:15` refers to the line and column of the rewrite rule `case`, is a local context synthesized by Squid (akin to \top_0 in Section 3.9.1) that is only valid within the pattern matching branch. This context is refined with `{y:Int}`, the context requirement of the rewritten term `pgrm`.

For instance, in:

```
val pgrm = code"val x: Int = ?y ; println(x + ?y)"

pgrm rewrite {
  case code"($n: Int) + ($m: Int)"
    => code"(-$m - $n) * (?z : Int)"
}
```

The return type of the `rewrite` expression above is `Code[Int, {y: Int; z: Int}]` because in the right-hand side of the rewrite rule, we return a term with context wider than that of its pattern, having extended it with some new free variable `z` — this context is inferred as `<context @ 2:15>{y: Int; z: Int}`.

This mechanism is essentially the same as the way rule T-T-RWR in Figure 6.3 expects the body of the rewriting to contain context brand κ , and removes that brand from the final result.

Run and closed terms. Squid’s `run` method is type-safe, as it statically rejects the evaluation of code that potentially contains free variables. This is achieved by making `run` take an implicit parameter which acts like an evidence [Oliveira et al., 2010] that the context of the term being run is the empty context. We reproduce the signature of `run` below, as it appears as part of the `Code[+Typ, -Ctx]` class:

```
type Code[+Typ, -Ctx] <: {
  def run (implicit ev: Ctx == {}): Typ
  ...
}
```

The implicit parameter `ev` expresses a requirement for an *evidence* that `Ctx` be the empty context `{}`. Evidence of the form `A == B` are generated by Scala’s standard library when the subtyping relations `A <: B` and `B <: A` are satisfied. As a result, it is impossible to call `.run` on a term that is not closed. For example, `code"?x: Int".run` results in a compilation error reading “cannot prove that `{x: Int} == {}`,” while the expression `code"val x = 123; ${`

`code"println(?x)"}.run` compiles¹⁴ and prints 123 to the console.

6.5 Application: Query Compilation By Rewriting

In this chapter, we discuss a very practically relevant application of metaprogramming techniques: query compilation, which is currently an active area of database research.

We have built a number of query compilers over the past years, including DBToaster [Ahmad and Koch, 2009, Koch et al., 2014] and LegoBase [Klonatos et al., 2014, Shaikhha et al., 2016], which had their part in starting and accelerating this trend. Building these systems required substantial effort, due to the need for generating low-level database code with state-of-the-art performance from queries expressed in complex high-level languages (like SQL).

Most existing query compilers are difficult to maintain because they work by basic template expansion, generating all the code in a single pass. To better separate the concerns of achieving advanced code optimization, one needs to design several independent transformation passes corresponding to different levels of abstraction [Shaikhha et al., 2016]. These passes should be statically type- and scope-checked to avoid potential mistakes.

Squid was designed in part as an answer to the metaprogramming needs discovered while iterating over the designs of these compilers. Early versions of Squid quasiquotes were used as part of real systems such as LegoBase, but to best explain the kinds of transformations used in our systems, we have designed a simpler, stripped down query compiler built entirely with Squid, available online,¹⁵ and presented in this chapter.

6.5.1 Systems as multi-level DSLs

Our general approach is to structure systems as multi-layer domain-specific languages (DSL): we start from a very high-level and declarative language on which powerful algebraic transformations can be performed, and then progressively *lower* the level of abstraction by removing higher-level constructs and adding more and more imperative ones, until we end up with optimal low-level code comparable to what an expert human programmer would have written directly [Shaikhha et al., 2016, Parreaux et al., 2017a].

In the rest of this chapter, we describe two of the central transformations of a query compiler implemented with Squid: schema specialization and row-to-column store transformation.

¹⁴Without a type annotation, free variable `x` is inferred to be of type `Any` — the type expected by `println`.

¹⁵This example and others can be found on the Squid-examples open source repository: <https://github.com/LPTK/squid-examples>.

6.5.2 Schema Specialization

Relational databases work by keeping some metadata (called the data dictionary) that represents what type of data is stored and its relation with the logical schema of the database. In a classical database system, this metadata is processed at runtime to determine how the data should be accessed and modified, given a high-level logical specification obtained from a query. This incurs high interpretive overhead, as it means that schema information has to be read over and over again, resulting in much repeated work, and that data accesses have to go through indirection.

The goal of the Schema Specialization transformer is to *optimize* or *stage away* all this overhead, specializing a query program to the current schema of the database (once it stops changing) and removing most of the indirection that would normally happen at query evaluation time. This transformer works similarly to partial evaluation, where the speculative rewrite rules of Squid are used as some form of dynamic *binding time analysis* [Jones et al., 1993], to extract the static parts from arbitrary programs. For example, we specialize query programs that use schemas expressed as lists of field information and completely remove that list data structure from the residual programs.

Consider the following data structures, used as the basis of a high-level query execution engine. To simplify the presentation, we assume that all columns of the relation are of type `String`. (In practice, modular abstraction with Scala path-dependent types can be used to abstract over the types, see `CodeType` in Section 1.3.3.) Instances of class `Row` internally store a list of column values, and instances of `Schema` store the list of the names associated to each of these columns:

```
class Row(values: List[String], size: Int) { ... }  
class Schema(columnNames: List[String]) { ... }
```

We want to transform a query program such as:

```
val s0 = new Schema("name", "age")  
val q = Relation.scan("data.csv", s0).project(Schema("age"))  
q.print
```

into a program where the `scan`, `project` and `print` methods are inlined to their underlying loop structures, which use methods `Row.getField` and `Schema.indicesOf`, so that we can then remove the schema data structure entirely. In the excerpt below, we show one particular rule of the schema specializer:

```

pgrm fix_rewrite {
  // $colNames* extracts a variable number of arguments as a sequence of terms
  case code"val s = new Schema(List($colNames*)); $body"
    => (body fix_rewrite {
      case code"s.columnNames"
        => code"List($colNames*)"
      case code"($r: Row).getField(s, $name)"
        if colNames.contains(name)
        => val index = colNames.indexOf(name)
           code"$r.getValue({Const(index)})"
      case code"s.indicesOf(List[String]($colNames2*))"
        => val columnIndexMap = colNames.zipWithIndex.toMap
           val indices = colNames2.map(columnIndexMap).map(Const)
           code"List($indices*)"
    }).subst["s"](abort())
}

```

The asterisk at the end of `$colNames*` indicates that we are extracting a variable list of arguments, giving `colNames` type `Seq[Code[String, _]]` (omitting it, we would match a single argument).

After this transformer is applied, we execute a general-purpose `List` partial evaluator (also written using `Squid`) to remove all schema indirections from the program. The following transformers in the pipeline of our query compiler then transform collections of rows (which are internally backed by a `List` of fields) into collections of tuples (which provide faster access to their components), with calls to `row.getValue(i)` with a constant index `i` are converted into tuple accesses.

6.5.3 Row-to-Column Store Transformer

Classical relational database systems such as IBM DB2, Oracle, and Microsoft SQL Server are “row-stores,” meaning that they store all their data records one after the other in memory. However, many recent systems, such as Vertica, SAP HANA, and others, have experimented with a “column-store” system where, for each fields of the records of a particular table, a separate storage structure is used — column stores are a very prominent research topic in databases, starting with C-store [Stonebraker et al., 2005]. Each approach has pros and cons, but database systems are currently either developed one way or the other, with no way to reconfigure them after the fact.

In our previous work [Klonatos et al., 2014, Shaikhha et al., 2016], we showed how to automatically translate one kind of system to the other. `Squid` makes that transformation type-safe (i.e., more robust) and much easier to express, as we saw in Section 6.2.5 — in essence, the array-

of-structs to struct-of-arrays optimization shown in Figure 6.1 corresponds to the row-store to column-store transformation of databases, when expressed in the context of in-memory query compilation.

6.6 Related Work

We now review some related work.

Previous Squid implementations

In its original implementation (presented in Chapter 1), the Squid type-safe metaprogramming framework provided statically-typed quasiquotes, but with more limited pattern matching capabilities. The only way to match bindings was to use higher-order pattern variables (similar to what was proposed by Sheard et al. [1999]), which means that matching the body of a binding construct necessarily resulted in a function term, so one could never really separate open code from its enclosing binding.

This posed problems when one wanted to change the nature of a binding (such as what happens in Figure 6.1) or when one wanted to open a binding, explore its body, and drive the reconstruction of that binding based on information gathered from the body — indeed, the reconstruction of the binding structure had to be set up before we could actually see the body.

Squid was used to enable quoted staged rewriting, an approach to library-defined optimizations (Chapter 4); in that work, we needed to work around these limitations and used an unsafe ‘close’ function to temporarily treat some code function as an open term (Figure 5.3). Misuses of that construct could lead to scope extrusion problems.

In addition, in the non-contextual version of Squid, users could call `.run` on arbitrary pieces of code, including open terms; this would result in runtime crashes.

In contrast, Squid’s new contextual quasiquote system, presented in this chapter, allows for very flexible binding analysis and reconstruction, while statically preventing scope extrusion and unbound variable reference errors.

Beyond more flexible pattern matching, we also found that expressing open terms using explicit free variables was a useful metaprogramming technique in its own right. For example, it allows for more relaxed multi-stage programming patterns, as noted by Kim et al. [2006]. This technique would not be type-safe without contextual quasiquotes.

Multi-stage formal calculi

Numerous multi-stage calculi based on modal logic have been developed that relate to our approach, including λ^\square [Davies and Pfenning, 2001] and λ° [Davies, 1996], which inspired

the design of MetaML. To prevent the evaluation of open code, Taha and Nielsen [2003] mention the possibility of reflecting context requirements in the type of terms but choose the more lightweight approach of environment classifiers, which unfortunately does not prevent imperative effects from causing scope extrusion. The systems by Nanevski [2002], Kim et al. [2006], as well as $\lambda^{\text{[]}}$ by Rhiger [2012a, 2005] use the contextual approach and do not have this problem. This approach was later given a foundational treatment by Nanevski et al. [2008], who presented an intuitionistic modal logic of necessity and its proof theory, and from this logic develop a contextual modal type theory, showing how modalities of necessity map to contexts. They discuss this type theory in the contexts of staging and logical frameworks. While we created our formal system by abstracting from the practical considerations of Squid pointed at throughout this thesis, the type-theoretic development carried over from that line of work turns out to be strongly analogous. Most notably, the ν^{\square} calculus by Nanevski [2002] presents code pattern matching using higher-order pattern variables (similar to what we used in Chapter 1), along with support for first-class manipulation of names (analogous to Section 7.5; but not formalized in $\lambda^{\text{[]}}$). In contrast, our calculus is not limited to two stages, allows for more flexible patterns that can match free variables, and lets pattern variables implicitly capture their local context. This gives us a simpler, yet more expressive account of code pattern matching. Furthermore, we allow the hygienic rewriting of all subterms of a code value at arbitrary depths¹⁶ (the **rewrite** construct), unlocking the power of speculative rewrite rules.

It is worth noting that environment classifiers were eventually replaced by runtime checks in the main MetaOCaml implementation because they gave “*good protection (a type error) against only rare errors, while being cumbersome always*” [Kiselyov, 2017]. They also gave relatively unhelpful error messages such as “error: ‘a not generalizable in (‘a, int) code,” while in Squid context errors manifest as understandable subtyping violations. Nevertheless, the problems of environment classifiers with mutable references were eventually solved via *refined* environment classifiers by Kiselyov et al. [2016], who gave a nice intuition on why using partially-ordered type variables is sufficient to solve the same problems as e.g., Rhiger [2012a]. However, whether refined environment classifiers can be extended to reason about pattern matching is an open question.

Interaction with cross-stage persistence

I noticed that in some cases where MetaML requires cross-stage persistence (CSP), we eschew it thanks to the use of non-lexically-scoped free variables (or explicit free variables, in Squid). For example, consider the program `<fun x → ~(run <⟨x⟩ >>)>` which in [Taha and Nielsen, 2003] requires classifier annotations (written ‘[.]’) and CSP annotations (written ‘%’), as in:

```
<fun x → ~((run (a) <%⟨x⟩>)[b])>
```

¹⁶Note that **rewrite** cannot be *encoded* with pattern matching in ν^{\square} (or in $\lambda^{\text{[]}}$) as that would require polymorphic recursion.

This program can be written without any notion of CSP or classifiers in λ^{B} as:

`[λx :Int. [run [[x]]]]`

and in Squid as `code"(x: Int) => ${ code{code"?x:Int"}.run }"` (where `code{...}` is an alternative syntax for `code"..."` which helps with nested quotations — see Section 3.9.2).

Query Compilation

Query compilation has been employed in database systems since the dawn of the relational database era: the very first relational database system, IBM’s System R, used query compilation in its early prototypes, but this approach was quickly abandoned in favor of query *interpretation*. Chamberlin et al. [1981] explain that this was ultimately due to the impracticality of writing and maintaining code generators for query engines, rather than the query engine code itself, in this early time of databases, when architectures and algorithms were still very much in flux and subject to experimentation. What is not explicitly stated there, though very clear, is that modern metaprogramming would have helped making the construction of query compilers much more manageable and sustainable.

Recently, also thanks to advances in programming languages and technologies such as LLVM, query compilation has returned to the limelight of databases, with commercial systems such as StreamBase, IBM Spade, Microsoft’s Hekaton, Cloudera Impala, and MemSQL employing it. Academic research has also intensified [Ahmad and Koch, 2009, Koch, 2010, Krikellas et al., 2010, Neumann, 2011, Koch, 2014, Koch et al., 2014, Klonatos et al., 2014, Viglas et al., 2014, Crotty et al., 2015, Rompf and Amin, 2015b, Nagel et al., 2014, Karpathiotakis et al., 2015, Armbrust et al., 2015].

6.7 Conclusion

In this chapter, we showed how to implement safe non-lexically-scoped open code manipulation for Squid. We formalized the approach as λ^{B} , a multi-stage calculus with pattern matching on code values that allows safe scope extrusion and rewriting of open code. We introduced “speculative rewrite rules,” an important class of type-safe optimizations based on the flexible manipulation of variable bindings. As an application example for speculative rewrite rules, we showed how to implement several query compiler optimization techniques inspired by real-world use cases.

7 Hygienic Scope Polymorphism

In Chapter 6, we demonstrated λ^{H} , a system for type- and scope-safe non-lexically-scoped open code manipulation. This relied on reflecting the context requirements of quoted program fragments in their types.

λ^{H} was sufficient for expressing advanced program optimizations based on program transformation primitives such as the `rewrite` construct. These primitives, which are careful to avoid name clashes by refreshing variable names as bindings are traversed, provide some limited form of safe & hygienic scope polymorphism: the ability, for a single piece of metaprogram, to manipulate programs in different contexts.

However, the λ^{H} system lacked the ability to more explicitly abstract over contexts, outside of `rewrite`-like primitives. Abstracting over contexts via *scope-polymorphic functions* is an important capability, offering more control on program transformations, enabling more metaprogramming code reuse, and allowing for more flexible open code manipulations in general.

In this chapter, we study how to extend the ideas of λ^{H} (and its realization in Squid) to supporting fully-general scope polymorphism, all the while preserving the safety and hygiene properties of λ^{H} .

We start by characterizing our notion of hygienic metaprogramming in the context of Squid (which is quite different from the usual notion of hygiene as found in the Lisp literature, for example), then show a first, negative result about adding safe and hygienic context polymorphism to λ^{H} with minimal changes, and finally demonstrate the $\lambda^{[\alpha]}$ calculus (along with its Squid implementation), a multi-stage language which achieves hygiene and context polymorphism by relying on first-class dependent affine binding representations.

The powerful stream fusion engine described in Chapter 5, which had to use unsafe escape hatches to cope with the strict lexical scoping imposed by earlier versions of Squid, was rewritten in this safer version of Squid with minimal changes.

7.1 Introduction

In Chapter 6, we introduced Contextual Squid and its formalization λ^{f} , with novel support for type- and scope-safe non-lexically-scoped open code manipulation. We presently review the limitations of that approach, and how we intend to resolve them.

The meaning of hygiene in this chapter

The notion of *hygiene* pervades the field of metaprogramming; this word is often used to describe various properties related to the sound handling of names and bindings in metaprograms. This usage is particularly prevalent in Scheme, but what it means exactly has proven elusive [Herman, 2010]. Similar notions of hygiene also naturally appear in other domains, such as contextual modal type theory [Nanevski et al., 2008].

We found from experience that the biggest problems encountered by Squid users with earlier version of Squid were scope extrusion (i.e., lack of *scope safety*) and unexpected shadowing — which we refer to as a lack of *hygiene*. This notion of hygiene is quite different from the traditional notion found in the Scheme literature, but it is neither easier nor harder to ensure; in fact, we argue that it occupies a separate (non-orthogonal) dimension of the design space.

First, Squid being a high-level metaprogramming system, we are not especially concerned with the necessity of ensuring fresh names — this necessity is satisfied quite trivially by construction (see the previous chapter), and has never cause problems in practice. Moreover, we are faced with different problems than the Scheme community: On the one hand, in Scheme, many of the difficulties related to hygiene come from the fact that macros can define their own arbitrary binding forms, which is not possible in Squid. On the one hand, we would like to ensure a stronger notion of name hygiene than in Scheme, not on the level of *macro expansion steps*, but rather on the level of arbitrary higher-order metaprograms. It is worth emphasizing here that macros are essentially *first-order* metaprograms, for which it is possible to define successive expansion steps, and around which a weaker notion of hygiene can be defined, but this is no longer possible in the general framework of Squid.

Therefore, for the purpose of this chapter, we will define hygiene as the absence of shadowing between identifiers in the generated programs, and we will argue that our final system preserves this property. Importantly, many shortcomings one could see in this definition of hygiene are actually handled by the separate notions of *scope safety* and *type safety*; we believe that these three properties together effectively make the system well-behaved, avoiding sources of unsoundness and surprises.

A simplistic approach to hygiene and scope safety

λ^{f} avoided any possible name clashes by carefully refreshing *bound* variable names as they were traversed by the `rewrite` primitive, and by statically preventing the shadowing of *free*

variables, which was possible because free variables appear in the types of program fragments. For instance, in a pattern matching or rewrite expression, it was not possible to pick a variable name, in the pattern, which clashed with the name of a free variable of the scrutinee:

```
def test(pgrm: Code[T, {x: Int}]) = pgrm rewrite {
  case code"(x: String) => $body" // error: name 'x' clashes with a FV in pgrm
    => ...
}
```

While this approach works well in this restricted setting, it becomes insufficient when considering more advanced forms of context polymorphism, as we will see.

Scope-polymorphic functions

Scope-polymorphic functions are functions that abstract over some context C , and can thus be applied to terms in different contexts, in a parametric way.

For example, consider the motivating example of Section 6.2.6; its signature was:

```
def optimize[T](pgrm: Code[T, {}]): Code[T, {}] = pgrm rewrite ...
```

Currently, this function can only be applied to closed programs. We may want to change the signature of the function as in:

```
def optimize[T, C](pgrm: Code[T, C]): Code[T, C] = pgrm rewrite ...
```

so that it can be applied on open program fragments of arbitrary context C . However, how should we make sure that this new definition will not mix up the free variable of C with the free variables introduced during the execution of the optimizer?

Remember that the body of `optimize` looked like:

```
pgrm rewrite {
  case code"val arr = ...; $body" =>
    val a = code"?a : Array[$ta]"
    ...
    code"val a = new Array[$ta]($size); ..."
}
```

where we introduce match a variable binding, giving it the name `arr`, and then introduce free variables `a` and `b` temporarily, in the scope of the rewrite rule.

If we are not careful, calling a polymorphic version of this optimizer on a program which contains free variables of these names, as in:

```
optimize[{ a: String }](
  code"val xs = new Array[(Int, Int)](42); (?a: String).length")
```

Chapter 7. Hygienic Scope Polymorphism

will result in the generation of ill-typed code:

```
code"val a = new Array[Int](42); val b = new Array[Int](42); a.length"
```

because the free variable `?a: String` in the original program will have been captured unintentionally by the binding introduced by the optimizer.

Context disjointness evidence

One way to prevent this kind of name clashes is to require the *absence* of variable names `arr`, `a`, and `b` from `C`, which can be expressed in Scala's type system by way of implicit evidence parameters.

We can make Squid disallow the refinements of abstract contexts unless these contexts are known not to contain the refined names. More generally, the rule would be that when type checking quasiquotes, whenever an abstract context `C` is intersected with any other context `D`, an implicit *disjointness* evidence of type `C <> D` is searched for. If no such evidence is found, the quasiquotation fails.

Type `<>[N, C]` is a simple type class with a private constructor, so that Squid only can create instances of it. All evidence of type `A <> B` are generated automatically for all appropriate concrete contexts by an implicit Scala macro, that checks that `A` and `B` share no common field names. Other instances are obtained by composition of implicit assumptions.

This way, our proposed definition of `optimize[T, C]` now fails to compile. Instead, one has to write the following declaration:

```
def optimize[T, C] (pgrm: Code[T, C])
    (implicit ev: C <> {arr: Any; a: Any; b: Any})
    : Code[T, C] = ...
```

Note that the types of the variables specified in context disjointness evidence are not important, so for instance `C <> {s: Any}` and `C <> {s: String}` are interchangeable.

With this new definition, we can no longer call `optimize` on the program which caused a name clash: the call is rejected with a compile-time error that reads:

```
"Cannot prove that {a: String} <> {arr: Any; a: Any; b: Any}."
```

Context disjointness is insufficient

Although the approach is technically sound, requiring context disjointness evidence every time one want to weaken an abstract context is too restrictive and greatly limits the usefulness of scope polymorphism. More specifically, this restriction:

- Hampers the modularity of code manipulations, violating the encapsulation of implementation details: if two transformers t_0 and t_1 want to locally use the same free variable names, then t_1 may not be able to be used in the body of t_0 , and conversely.
- Prevents the definition of polymorphically-recursive metaprograms,¹ which need to pass refined contexts in each recursive call.

The latter point is of central importance, as polymorphically-recursive functions are common in statically-typed metaprogramming where contexts are encoded in the types of code values. It is often necessary to define recursive functions that introduce new variables into a context on each recursive call, and these names should not conflict [Nanevski, 2002].

Example 7.1.1. *Consider the following function, which recursively creates a list of bindings of every natural number until n , and then sums them up all together.²*

```
def sumTo[C](i: Int, n: Int, body: Code[Int, C]): Code[Int, C] =
  if (i <= n)
    code"val x = ${Const(i)}; ${ sumTo(i + 1, n, code"$body + (?x: Int)" ) }"
  else body
```

Evidently, we should have a way to hygienically generate each binding for x so that they do not conflict — otherwise, we would end up with executions like `sumTo(1, 2, code"0") == code"val x = 1; val x = 2; 0 + x + x"`, which is not the intended result.

In Squid, it is also common to see transformers which are defined recursively, i.e., where the transformer calls itself, in the right-hand side of one of its pattern matching or rewriting expressions, on terms which have been locally extruded from their context. This is the case, for instance, of the `flatMap` streamlining transformer presented in Chapter 5 (Figure 5.3). There again, we should make sure that the extruded names do not conflict across recursive calls.

Example 7.1.2. *Consider the following function, which recursively decomposes a list of bindings, adding logging statements on each of them:*

```
def logLocalVars[T, C]: Code[T, C] => Code[T, C] = {
  case code"val x: $xt = $xv; $body"
    => code"val x = $xv; println(x); ${ logLocalVars(body) }"
  case code"$e; $body"
    => code"$e; ${ logLocalVars(body) }"
  case _ => body
}
```

¹Polymorphic recursion happens when the recursive calls of a function take different type arguments than in the current call.

²This is a contrived example, which would be better solved with a cross-quotation reference; we only give it for clear illustration purposes.

Again, such a transformation should avoid conflating the different variables extruded in each recursive call by inadvertently renaming them all to `x`.

Research questions

The specific research questions we tackle in this chapter are the following:

- How to characterize binding hygiene in the context of a general-purpose non-lexical metaprogramming framework like Squid?
- Can we refine the dynamic semantics of context polymorphism to make it sound, in the absence of disjointness requirements?
- How to design a system for handling bindings like first-class entities while preserving scope safety and hygiene?

7.2 Metaprogramming Hygiene Beyond Macros

Hygiene has been traditionally defined in terms of “*transcription steps*” [Kohlbecker et al., 1986], i.e., macro expansion stages. In his formalization of the essence hygiene in the context of Scheme, a Lisp dialect, Adams [2015] noted that:

hygiene is essentially lexical scoping at the macro level

However, in the setting of Squid metaprogramming, code manipulation is not organized around macros and does not follow obvious expansion steps, and moreover open code manipulation is not even lexically-scoped.³ This is a major difference: while it is possible to clearly define macro input and macro output, as macros are essentially *first order* (syntax to syntax), that distinction is no longer possible in a general metaprogramming framework like Squid, which can freely accept functions and return functions, so that there is no easily discernable syntax-to-syntax expansion steps involved.⁴

So what exactly do we mean by hygiene, and how can it be obtained?

We have already started fleshing out an understanding of hygiene in Section 1.2.3, separating the properties of a hygienic system into two parts:

³Note that on the flip side, hygiene in Lisp considerably complicated by the fact that Lisp macros can *create new binding forms* and manipulate program fragments which are not yet well-formed (they expand outside-in and manipulate program fragments whose binding structures are not yet fully defined), capabilities that we are not interested in supporting in Squid.

⁴Seen another way: previous work focus on the macro boundary; they do not do anything to prevent local macro helper functions from doing unhygienic manipulations (one local function may introduce a binding inadvertently captured by another). There is no *hygienic quasiquotation* system in Lisp dialects, to the best of my knowledge.

- *reference hygiene*, which is the property that references created within some metaprogram will not get captured by *unrelated* binders in the manipulated program; and
- *binder hygiene*, which is the property that binders introduced by some metaprogram will not capture *unrelated* pre-existing identifiers in the manipulated program.

The key word in both of these properties is the word *unrelated*. How shall we decide what is and is not related?

Lexical and non-lexical scoping

In a lexically-scoped metaprogramming system, all variable references statically know their binders, which are in the surrounding typing environment. Therefore, it is easy to apply a renaming on all the binders in a program at compilation time, ensuring that “unrelated bindings” do not interfere — this is what is done in traditional multi-stage programming systems like MetaML [Taha and Sheard, 2000, Kiselyov, 2014, Kiselyov et al., 2016].

On the other hand, in a non-lexically-scoped system like Contextual Squid, it is generally not possible to statically determine a single binder for each variable reference, as a variable occurrence in a given program may end up being bound by *different* binders and different places in the metaprogram, in scopes which are not even visible from the context of the variable reference.

To be more concrete, consider that a library may expose some open program fragments such as the following one:

```
object MyLibrary {
  val cde: Code[Int, {v: Int}] =
    code"(?v: Int) + 1"
}
```

It is evidently not possible to know statically all the places where this free variable *v* will be bound, and so a global renaming scheme will not work.

7.2.1 Hygiene Via the Type System

The idea we introduce in this subsection is that hygiene in the presence of non-lexical scoping could be recovered through the type system.⁵

Since a type system like Squid’s keeps track of all the free variables present in each program fragment, it should be possible to statically determine what variable occurrences are “related” or “unrelated,” i.e., *known* or *unknown* in each metaprogramming context.

⁵This is not a new idea. Herman and Wand [2008] already considered the use of types to precisely specify hygiene; however, this was for a very different, Lisp-like system.

For example, consider the following program:

```
def foo[C]: Code[Double, C] => Code[Double, C] = {  
  case code"math.pow(x, 0)" => code"1.0"  
  case code"math.pow(x, 2)" => code"x * x"  
  case ...  
}
```

which should not type check, because no `x` is statically known to live inside `C`.

On the other hand, the following program should type check:

```
def foo[C]: Code[Double, C { x: Double }] => Code[Double, C { x: Double }] = {  
  case code"math.pow(x, 0)" => code"1.0"  
  case code"math.pow(x, 2)" => code"x * x"  
  case ...  
}
```

As an aside, note that the *variable matching* syntax used above should be distinguished from the *subterm extraction* syntax, also valid, but with a different semantics (it will extract any term, not a specific variable):

```
def foo[C]: Code[Double, C] => Code[Double, C] = {  
  case code"math.pow($_, 0)" => code"1.0"  
  case code"math.pow($c, 2)" => code"$c * $c"  
  case ...  
}
```

7.2.2 Naive Interpretation of Context Polymorphism

We have seen in the introduction that naive interpretation of context polymorphism (or *scope* polymorphism) leads to unsoundness. To give a simpler example of this phenomenon, consider the context-parametric functions `ref` and `bind` below:⁶

```
def ref [C](n: Code[Int, C]) = code"(?s: String).take($n)"  
def bind [C](m: Code[String, C { s: String }]) = code"(s: String) => $m"
```

The return type inferred for `ref` is `Code[String, C { s: String }]`, because the context requirement `C` introduced by `n` is propagated to the main term, but that context is extended with the new requirement for a free variable `s` of type `String`, introduced by the `(?s: String)` free variable syntax. The return type inferred for `bind` is `Code[String => String, C]`, because the `s` variable in the context requirement of `m` is captured by the lambda abstraction.

⁶Note that in Scala, `str.take(n)` represents the `n` first characters of a string `str`, or `str` if `str.length < n`. This method is added via an implicit conversion, but our quasiquotes allow us to ignore it completely.

The following REPL session demonstrates unproblematic usages of these definitions:

```

val a = code"?x : Int"
      : Code[Int, {x: Int}]
      → code"?x"

val b = ref(a)
      : Code[Int, {s: String; x: Int}]
      → code"?s.take(?x)"

val c = bind[{x: Int}](b)
      : Code[String => String, {x: Int}]
      → code"(s: String) => s.take(?x)"

code"val x = 12; $c"
      : Code[String => String, {}]
      → code"x = 12; (s: String) => s.take(x)"

```

A problem with context polymorphism as presented above arises when an abstract context which gets refined at some point is instantiated with a concrete type that is incompatible with such refinement.

As an example, the result type of `ref(code"?s : Int")` is the problematic structural type `{s: Int & String}`. This type is simply not a realizable context, making the result of such a call unusable. A subtler problem arises when we refine a context viewed as abstract with a variable that it already contains, *and capture this variable* before the context type is concretized. An example of this can be composed with the `ref` and `bind` seen above:

```

def compose[C](x: Code[Int, C]) = bind(ref(x))

compose(code"?s : Int")
      : Code[String => String, {s: Int}]
      → code"(s: String) => s.take(s)"

```

Observe that the result `code"(s: String) => s.take(s)"` is ill-typed! The problem is that we introduce a mismatch between the static semantics of contexts, handled by quasiquotes at compilation time, and the dynamic semantics of free variables.

This is fundamentally a hygiene problem: the type safety violation arises from a violation of hygiene. Using intensional type analysis (Section 6.3.3), we could change the substitution rules to avoid substitutions which result in type errors, solving type safety *per se*, but such a system would be deeply unsatisfactory, as it would still be fundamentally unhygienic and un-parametric (variables would behave differently depending on the types of the variables being bound).

7.3 A Negative Result: No Hygiene With Plain Names

This section is concerned with the question of whether we can refine the dynamic semantics of quasiquotation in λ^{B} and Squid to make it conform to the obvious static semantics of context polymorphism. This way, we could ensuring hygiene through the type system without having to deeply alter the user-visible language.

In other words, the central idea we pursue (unsuccessfully) in this section is *to enforce hygiene by leveraging the type system, in the presence of non-lexically-scoped open terms, scope polymorphism, and plain variable names*.

This is a surprisingly subtle problem on which I have spent a significant amount of time, in vain, as the problem seems not to have any satisfactory solutions: while some solutions *seem* to work at first, and can handle *most* common usages flawlessly, formal analysis demonstrates that they always fall apart in corner cases. Again, what makes the problem particularly tricky is the fundamentally higher-order nature of our language (to be contrasted with the first-order problem of hygiene in Lisp macros).

Although it is possible to define nonstandard operational semantics to achieve the stated goal of this section, these semantics are not compatible with existing language implementation techniques, and thus the resulting systems cannot be easily embedded into Scala and Squid.

This section is only interesting insofar as it provides a strong (if informal) argument why hygiene with plain names and non-lexical open code manipulation with context polymorphism is *not* feasible. ***Readers who are not interested in this negative result may safely skip this section, and proceed to the next one (Section 7.4).***

7.3.1 Core Problem

Formally, we are looking to add the following two simple rules to the type system of λ^{B} :

$$\begin{array}{c} \text{T-CTXABS} \\ \frac{\Gamma_0, \Gamma \cup \{\kappa\} \vdash t : T}{\Gamma_0, \Gamma \vdash \Lambda \kappa. t : \forall \kappa. T} \\ \text{T-CTXAPP} \\ \frac{\Gamma_0, \Gamma \vdash t : \forall \kappa. T}{\Gamma_0, \Gamma \vdash t[C] : [\kappa \mapsto C]T} \end{array}$$

where κ is a context variable which can be abstracted with T-CTXABS and then substituted with a concrete context with T-CTXAPP.

The core problem of this chapter is that a naive context substitution approach may leave some *terms* ill-typed as a result of unintended variable capture. For example, consider the following term:

$$\begin{aligned} t &: \forall \kappa. \text{Code Int } \{\kappa\} \rightarrow \text{Code (Bool} \rightarrow \text{Bool)} \{\kappa\} \\ t &= \Lambda \kappa. \lambda c : \text{Code Int } \kappa. [\lambda x : \text{Bool}. [c]] \end{aligned}$$

Now consider what happens when we apply t as in $t[\{x:\text{Int}\}]$, passing $\kappa = \{x:\text{Int}\}$ and $c = [x]$. The typing rule T-CAPP tells us that the result type is:

$$[\kappa \mapsto \{x:\text{Int}\}](\text{Code } (\text{Bool} \rightarrow \text{Bool}) \{ \kappa \}) = \text{Code } (\text{Bool} \rightarrow \text{Bool}) \{ x:\text{Int} \}$$

This in itself is fine. But if we applied context substitution on the term in the same way as on the type, we would end with the term:

$$[\kappa \mapsto \{x:\text{Int}\}](\lambda c:\text{Code Int } \kappa. [\lambda x:\text{Bool}. [c]]) = \lambda c:\text{Code Int } \{x:\text{Int}\}. [\lambda x:\text{Bool}. [c]]$$

which is no longer well-typed! Indeed the free variable x in c now clashes with the locally-defined $\lambda x:\text{Bool}$ in the term.

And indeed, if we tried to evaluate the term:

$$\text{run } [\lambda x:\text{Int}. [t[\{x:\text{Int}\}][x+1]]]$$

using the evaluation rules of Figure 7.3, we would get stuck while trying to run the code value:

$$[\lambda x:\text{Int}. \lambda x_0:\text{Bool}. x_0 + 1]$$

Clearly, the dynamic semantics of context-polymorphic functions has to be revised, as a naive approach does not work.

There are many ways we could tackle the problem. For instance, we could change the output of context substitution so that instead of immediately replacing context variables κ with concrete contexts C , it would replace them with bindings $\kappa \mapsto C$ which would be considered *opaque* as long as we are evaluating terms *in the context* where κ was introduced.

7.3.2 Reified Context Parameters

The first observation we can make is that we need to reify some context representations one way or the other, in order to let the dynamic semantics of quasiquotes be influenced by the static context types.

Indeed, the key idea is to distinguish the composition of code whose context is *known*, which means that its free variables should be captured, from the composition of code with *unknown* context, whose free variables should be left alone for hygiene reasons. For this, we need to keep track of the contexts that are in scope when code manipulations are performed.

In Scala, reifying type parameters can be done with a type class (like the $A \lt; B$ type class mentioned in Section 7.1). We can define a `Ctx` type class which can be used to identify abstract contexts uniquely, as well as their composition via refinement and intersection. For instance, a recursive context-polymorphic function could be declared as:

```
def foo[C: Ctx, D: Ctx] = {  
  ... foo[C & D, D { v: Int }]  
}  
  
foo[{v: Int}, {w: String}]
```

In the example above, two new context evidence values would be created for the `foo[{v: Int}, {w: String}]` call, respectively for contexts `C = {v: Int}` and `D = {w: String}`, and these evidence values would be composed at runtime in order to create the evidence value passed into the recursive calls of `foo`. Conceptually, the program above, after elaboration, should look something like:

```
def foo[C, D](implicit C: Ctx[C], D: Ctx[D]) = {  
  ... foo[C & D, D { v: Int }](CtxEv.inter(C, D),  
    CtxEv.inter(D, new CtxEv[{v: Int}]))  
}  
  
foo[{v: Int}, {w: String}](new CtxEv[{v: Int}], new CtxEv[{w: String}])
```

Note: reified contexts should not lead to surprising semantics

Importantly, the reification of context parameters should only be used for hygiene considerations, and should not lead to a loss in parametricity.

Moreover, sometimes it is possible to infer different possible type arguments to satisfy a function call; in such context, type inference should not influence the dynamic semantics of the call. For example, in:

```
def foo[C: Ctx](a: Code[Int, C{ x: Int }]): Code[Int => Int, C] =  
  code"(x: Int) => $a"  
  
val r = foo[{}](code"(?x: Int) * 2")  
val s = foo[{x: Int}](code"(?x: Int) * 2")
```

Both calls are type-correct, and they should have the same semantics. If they did not, the dynamic semantic of our language would be influenced by the (sometimes arbitrary) choices made during type inference — not an uncommon thing in Church-style languages like Scala and Haskell, but also not desirable in this case.

7.3.3 Reified Weakening

The next observation is that we can no longer allow the contexts of code values to be weakened implicitly (by upcasting, leveraging a subtyping relation, as in Chapter 6).

The need for reified weakening is most evident in the following example, where the same term is viewed under two different contexts in a context-polymorphic function, which *should* lead to different operational semantics in order to ensure hygiene:

```
def foo[C: Ctx](c0: Code[Int, C], c1: Code[Int, C { x: Int }])
  : Code[Int => Int, C]
  = code"(x: Int) => $c0 + $c1"

val c = code"(?x: Int) + 1"
val r = foo(c, c)
```

Note that after type inference, `foo(c, c)` has to be elaborated as `foo[{x: Int}](c, c)` for the call to type check.

According to our hygiene criteria, `r` should evaluate to `code"(x_0: Int) => ?x * 2 + x_0 * 2"` so that, for instance, `code"(x: Int) => $r"` evaluates to `code"(x_1: Int) => (x_0: Int) => x_1 * 2 + x_0 * 2"`. Indeed, `foo` should *not* capture the variable `x` which lives in `c0`, because that variable does not appear in its static context.

This can be achieved by noticing that in order to insert `c0` into the context of the lambda abstraction binding `(x: Int)` inside `foo`, we have to *weaken* its type from `Code[Int, C]` into `Code[Int, C {x: Int}]`. Reifying this weakening should allow us to make sure the dynamic semantic of quote insertion avoid substituting `x` in `c0`, since *in the context of* `foo`, `x` was not known to be free in `c0`.

In Scala, reified weakening can be implemented by relying on implicit conversions instead of a subtyping relation. With the weakening conversions applied explicitly, the definition of `foo` seen above becomes:

```
def foo[C: Ctx](c0: Code[Int, C], c1: Code[Int, C { x: Int }])
  : Code[Int => Int, C]
  = code"(x: Int) => ${c0.weaken[C { x: Int }]} + $c1"
```

Where a `c.weaken[X]` method call itself will capture a runtime representation of the reified context `X`, which will be inspected while evaluating code composition. The signature of `weaken` would be as follows:

```
type Code[+T, C] <: {
  def weaken[D <: C](implicit ev: C): Code[T, D]
  ...
}
```

Notice that `weaken` takes an implicit evidence for context `C` and not for `D` — we only care about the presence of free variables in `C`, the “true” context of the code value.

7.3.4 Context Evidence Opacity and Transparency

The crucial part of a hygienic operational semantics based on the ideas presented in this section is to know when a given reified context parameter is supposed to be *transparent* or *opaque*, while composing program fragments together.

Consider the following definition of `foo[C]`, where a weakening conversion to context `C { x: Int }` is reified around some code value `c`, and then the resulting code value is immediately returned:

```
def foo[C: Ctx]
  : Code[Int, C] => Code[Int, C { x: Int }]
  = c => c.weaken[C { x: Int }]
```

In the call site, we apply the function as `foo[{ x: Int }]`, which elaborates to something like `foo[{ x: Int }](C0)` where `C0 = new CtxEv[{ x: Int }]` is a fresh context evidence. We then insert the result of this call into a context where `x` is bound:

```
val c0 = foo[{ x: Int }] // passes fresh C0 evidence implicitly
val c1 = c0(code"?x: Int") // c1 has type Code[Int, { x: Int }]
code"(x: Int) => $c1"
```

The value we will get at runtime for `c1` will have the shape `code"?x: Int".weaken(C0)`, where `C0` is the context evidence which was synthesized for `foo`. In this call site context, `C0` is to be considered transparent, since we are no longer within its defining scope (which is the body of the `c0` closure) and we can therefore “see through” it, seeing that `c1` does indeed contain a genuine free variable `x`.

Compare that with a situation where the code composition would have happened in a context where `C0` was still opaque, and where we should therefore not have captured the free variable `x`, for example in:

```
def foo[C: Ctx]
  : Code[Int, C] => Code[Int => Int, C]
  = c => code"(x: Int) => ${c.weaken[C { x: Int }]}"
```



```
val c0 = foo[{ x: Int }] // passes fresh C0 evidence implicitly
val c1 = c0(code"?x: Int") // c1 has type Code[Int, { x: Int }]
assert(c0 == code"(x_0: Int) => ?x: Int")
```

In the example above, the context evidence `C0 = new CtxEv[{ x: Int }]` should be considered opaque while executing the code composition in `foo`, since `foo` is not supposed to see through the context parameter `C`.

7.3.5 A Problematic Program

The specific mechanism we should use to determine whether a given context is to be considered opaque or transparent is left undetermined,⁷ and is irrelevant. Indeed, in this subsection, we show with a counter example that *no matter which mechanism is used, that mechanism is necessarily wrong* if we assume “reasonable” operational semantics!

Consider the following program:

```
def foo[C: Ctx](c0: Code[Int], C)
  : (Code[Int, C { x: Int }], Code[Int, C { x: Int }]) => Code[Int, C]
= {
  val a: Code[Int, C { x: Int }] = c0.weaken
  val f: Code[Int, C { x: Int }]) => Code[Int => Int, C]
    = c1 => code"(x: Int) => $a + $c1"
  (f, a)
}

val c = code"?x: Int"
val (f, a) = foo[{ x: Int }](c)
// f: Code[Int, { x: Int }]) => Code[Int => Int, { x: Int }]
// a: Code[Int, { x: Int }]
f(c)
```

The program above *should* return `code"(x_0: Int) => ?x + x_0"`. Indeed, from the reasoning on hygiene we have developed, we can establish two facts on the code composition being performed inside the closure that is returned by `foo`:

- the `x` variable in `a` *should not* be captured by the lambda: `C` is opaque to `foo`, and therefore even after weakening `c0` to context `C { x: Int }`, `foo` should not be able to see the variable `x` that lives inside `C` — and therefore it should not capture it (this is necessary not only for hygiene but also for type safety, as explained in Section 7.2.2).
- the `x` variable in `c1` *should* be captured by the lambda: indeed, its static type has context `C { x: Int }` where `x` is visible without weakening; moreover, the variable comes from the call-site context where it had the completely unambiguous context `{ x: Int }`.

However, when the closure produced by `foo` executes in the program above, `c0` and `c1` are *exactly the same value*, simply passed in through two different paths. At no point do we have the opportunity to alter the value to reflect the difference in these paths: we cannot perform a special action on capture nor on argument passing, unless we accept some nonstandard

⁷We could, for example, pass additional implicit information along the context evidence in order to reify the call stack of the different context-polymorphic functions and determine whether, in the current execution context, a given context evidence is opaque or transparent.

operational semantics which is no longer aligned with Scala, or with most other existing programming language for that matter. So both `a` and `c1`, which are the same value in this example (and even have the same static type), should behave identically, which violates the bullet points above.

Therefore, *there is no reasonable operational semantics* for our language where the program above executes hygienically.

7.4 Hygiene Via Affine First-Class Bindings in $\lambda^{[\alpha]}$

We have seen in the previous section that using plain names in the context of λ^{\exists} and Squid simply *cannot* work. Therefore, we need to change something about the way we handle variables, in order to soundly support context polymorphism.

The main idea of this section is to use a first-class representation of bindings — which we refer to as *symbols* — to abstract over the name (and associated type) of variables in a program.

7.4.1 First-Class Bindings in Squid

We have added a `Variable[T]` data type to the Squid framework to represent variable bindings — also called *symbols* — as first-class entities which can be explicitly inserted into and extracted from programs.

The signature of this data type is as follows:

```
type Variable[T] <: {
  type C
  def toCode: Code[T, C]
  def tryClose[Ty, Co](pgrm: Code[Ty, Co & C]): Option[Code[Ty, Co]]
  def substitute[Ty, Co](pgrm: Code[Ty, Co & C], vlu: Code[T, Co]): Code[Ty, Co]
  ...
}
val Variable: {
  def apply[T]: Variable[T]
}
```

An instance of `Variable[T]` represents a free variable with a unique name as well as its associated type `T` (internally, Squid generates a fresh name on every instantiation). This is encoded by *each* instance having a *separate* type member `C` representing the context dependency associated with that symbol.

To keep track of symbol dependencies, we use Scala's support for path-dependent types [Amin et al., 2016] (as in Section 1.3.4 for type evidence): if `v0` has type `Variable[T]`, then `v0.C` refers

to the context dependency of *that specific symbol*, which is considered distinct from $v1.C$, for some other symbol $v1$, unless we can prove that $v0 == v1$.

One can obtain a reference to a free variable via its method `toCode` (or by directly inserting it into a program fragment), and we can try to (partially) close a program fragment `pgrm` in which the variable is statically known to potentially be free, by using method `close` (which corresponds to speculative closure, as in Chapter 6).

Examples

Recall Example 7.1.1, which inductively constructed a program fragments made of a succession of bindings. This example can now be implemented hygienically as:

```
def sumTo[C](i: Int, n: Int, body: Code[Int, C]): Code[Int, C] =
  if (i <= n) {
    val v = Variable[Int]
    code"val $v = ${Const(i)}; ${ sumTo(i + 1, n, code"$body + $v" ) }"
  } else body
```

In fact, *this is precisely how Squid implements cross-quotation references behind the scenes*. Indeed, the code below is automatically converted into the code above by Squid, when the Squid macros expand:

```
def sumTo[C](i: Int, n: Int, body: Code[Int, C]): Code[Int, C] = {
  if (i <= n)
    // cross-quotation reference of 'v':
    code"val v = ${Const(i)}; ${ sumTo(i + 1, n, code"$body + v" ) }"
  else body
}
```

In the examples above, the inner quote `code"$body+ $v"` or `code"$body+ v"` has type `Code[Int, C & v.C]`, and the context requirement $v.C$ is satisfied by binding present in the outer quote, so that the outer quote can be typed `Code[Int, C]`.

Also recall Example 7.1.2, which inductively decomposed a program, adding logging operations on each binding. That example can be rewritten as follows, by *extracting symbols* from the matched code patterns:

```
def logLocalVars[T, C]: Code[T, C] => Code[T, C] = {
  case code"val $x: $xt = $xv; $body" // notice the $x
    => code"val $x = $xv; println($x); ${ logLocalVars(body) }"
  case code"$e; $body"
    => code"$e; ${ logLocalVars(body) }"
  case _ => body
}
```

In the right-hand side of the case where we extract symbol `$x`, we get a variable `x` of type `Variable[xt.T]` in scope.

Enforcing Hygiene

The system as we have seen so far is scope-safe, but still not quite hygienic. As a simple example, recall Example 7.1.1 again, and consider that by misusing `Variable`, we could have easily generated programs containing (probably unintended) shadowing, a typical giveaway for the lack of hygiene:

```
val v = Variable[Int] // a single v is defined outside!
def sumTo[C](i: Int, n: Int, body: Code[Int, C]): Code[Int, C] =
  if (i <= n)
    code"val $v = ${Const(i)}; ${ sumTo(i + 1, n, code"$body + $v" ) }"
  else body
```

The program above will have the same unhygienic behavior as our first version in Section 7.1.

From this example, it is clear that some *linearity* (or more generally *affinity*) restrictions should be set in place to prevent the misuse of `Variable` symbols: indeed, in order to preserve hygiene, one should only be able to bind each variable symbol *at most once*.

7.4.2 Presentation of $\lambda^{[\alpha]}$

In this subsection, we present the $\lambda^{[\alpha]}$ multi-stage calculus, which reflects the augmentation of Squid with hygienic first-class symbol manipulation and context polymorphism.

Syntax

The syntax of $\lambda^{[\alpha]}$ is presented in Figure 7.1, where the new constructs (compared to $\lambda^{\{\}}$) are highlighted in grey.

The *scope abstraction* $\Lambda K. t$ and *scope application* $t[K]$ constructs can not only manipulate context variables $K = \kappa$, but also variable symbols $K = \alpha : T$. The corresponding function spaces are written $\forall K. T$.

	Term	$v ::=$	Value
$t ::= q_u$		n	<i>Literal</i>
$q_\theta ::=$	(Syntax Template)	$ \langle \lambda \underline{c} . t, \gamma \rangle$	<i>Closure</i>
n	<i>Literal</i>	$ \llbracket q_\emptyset \rrbracket$	<i>Quote</i>
$ t + t$	<i>Addition</i>		
$ x, y, z$	<i>Variable</i>		
$ t t$	<i>Application</i>	$\gamma ::= \{\overline{d}\}$	Subs. Context
$ \lambda x : T. t$	<i>Abstraction</i>	$d ::=$	
$ \Lambda K. t$	<i>Scope Abstraction</i>	$x \mapsto v$	
$ t[K]$	<i>Scope Application</i>	$ \alpha \mapsto [x : T]$	
$ \text{let } \alpha : T \text{ in } t$	<i>Symbol Creation</i>		
$ \llbracket q_u \rrbracket$	<i>Quote</i>	$T, S ::=$	Type
$ \text{run } t$	<i>Evaluation</i>	Int	<i>Integer</i>
$ t \text{ match } [t] \Rightarrow t \text{ else } t$	<i>Pattern Matching</i>	$ T \rightarrow T$	<i>Function</i>
$ t \text{ rewrite } [t] \Rightarrow t$	<i>Term Rewriting</i>	$ \forall K. T$	<i>Univ. Context</i>
$ \text{close}_\alpha t \text{ else } t$	<i>Speculative Closure</i>	$ \text{Code } T C$	<i>Code</i>
$ t : T$	<i>Type Ascription</i>		
$ \theta$	(Syntax Extension)	$C, \Gamma ::= \{\overline{c}\}$	Typ. Context
$u ::=$		$c ::=$	
$[x]$	<i>Variable Unquote</i>	$x : T$	<i>Binding</i>
$ \text{const } x]$	<i>Constant Unquote</i>	$ K$	<i>Abs. Context</i>
$ \llbracket \alpha \rrbracket$	<i>Symbol Unquote</i>	$K ::=$	
$ \lambda [\alpha]. t$	<i>Symbol Binding</i>	$\alpha : T$	<i>Symbol</i>
		$ \kappa, \alpha$	<i>Brand</i>

 Figure 7.1 – Syntax of $\lambda^{[\alpha]}$.

We have added antiquotation forms for inserting symbol references $[\alpha]$ as well as symbol bindings $\lambda[\alpha]. t$, and a way to create new symbols with **let** $\alpha : T$ **in** t .

Static semantics

The static semantics of $\lambda^{[\alpha]}$ is presented in Figure 7.2.

Rules T-SYMAPS and T-SYMAPP handle symbol abstraction and application, respectively. To enforce affinity, we use function $\#_\alpha(t)$, which counts the number of times symbol α is *consumed* (i.e., bound as part of a program fragment) in t :

$$\begin{aligned}
 \#_\alpha(x) &= \#_\alpha(n) = \#_\alpha([x]) = \#_\alpha([\mathbf{const} \ x]) &&= 0 \\
 \#_\alpha([\alpha']) &&&= 0 &&\text{even when } \alpha = \alpha' \\
 \#_\alpha([t]) &&&= \#_\alpha(t) \\
 \#_\alpha(t[\alpha]) &= \#_\alpha(\lambda[\alpha]. t) &&= 1 + \#_\alpha(t) \\
 \#_\alpha((\lambda x : T. t_0) t_1) &&&= \#_\alpha(t_0) + \#_\alpha(t_1) \\
 \#_\alpha(t_0 t_1) &&&= \#_\alpha(t_0) + \#_\alpha(t_1) \\
 \#_\alpha(\lambda x : T. t) &&&= \#_\alpha(t) \times 2 \\
 \#_\alpha(\Lambda K. t) &&&= \#_\alpha(t) &&\text{when } K \neq (\alpha : T) \\
 \#_\alpha(\Lambda \alpha : T. t) &&&= 0 \\
 \text{etc...} &&&(\text{other cases unsurprising})
 \end{aligned}$$

We consider the capture of a symbol into a function which consumes that symbol as an arbitrary number of uses of that symbol (we represent that as $\#_\alpha(t) \times 2$) — indeed, we do not have a way to prevent the capturing closure from executing more than once. Notice that we have a special case for let bindings (encoded as applied lambdas $(\lambda x : T. t_0) t_1$), since this is a common special-case where we can be sure that the closure is indeed executed only once.

Rule T-LETSYM handles symbol creation by reusing a judgement from T-ABSSYM and additionally making sure that the corresponding symbol α does not leak into the result type.

Rules T-CTXABS and T-CTXAPP handle context abstraction and application, respectively.

The side-condition $\bar{A}(x : S) \in C$ of T-CTXAPP and T-ANTI0 makes sure that contexts containing free *plain* variable names cannot be passed as argument (which could result in ill-typed terms, as explained in Section 7.3.1), and cannot be unquoted. Therefore, cross-quotation references with plain names are disallowed, but this is not a significant limitation: we can always encode something like $[\lambda x : T. \dots [x + 1] \dots]$ as **let** $\alpha : T$ **in** $[\lambda [\alpha]. \dots [[\alpha] + 1] \dots]$. This transformation can be applied automatically, which is essentially what the Squid implementation does to implement cross-quotation references.

Note that one may match normal lambda patterns only if they extract nothing from their

Term typing

$\frac{}{\Gamma_0, \Gamma \vdash n : \mathbf{Int}}$	$\frac{\text{T-PLUS} \quad \Gamma_0, \Gamma \vdash t_0 : \mathbf{Int} \quad \Gamma_0, \Gamma \vdash t_1 : \mathbf{Int}}{\Gamma_0, \Gamma \vdash t_0 + t_1 : \mathbf{Int}}$	$\frac{\text{T-ASC} \quad \Gamma_0, \Gamma \vdash t : T}{\Gamma_0, \Gamma \vdash (t : T) : T}$
$\frac{\text{T-ABS} \quad \Gamma_0, \Gamma \cup \{x : T\} \vdash t : S}{\Gamma_0, \Gamma \vdash (\lambda x : T. t) : T \rightarrow S}$	$\frac{\text{T-VAR} \quad (x : T) \in \Gamma}{\Gamma_0, \Gamma \vdash x : T}$	$\frac{\text{T-APP} \quad \Gamma_0, \Gamma \vdash t_f : T \rightarrow S \quad \Gamma_0, \Gamma \vdash t_a : T}{\Gamma_0, \Gamma \vdash t_f t_a : S}$
$\frac{\text{T-SYMAbs} \quad \Gamma_0, \Gamma \cup \{\alpha : S\} \vdash t : T \quad \#_\alpha(t) \leq 1}{\Gamma_0, \Gamma \vdash (\Lambda \alpha : S. t) : \forall \alpha : S. T}$		
$\frac{\text{T-SYMApP} \quad (\alpha' : S) \in \Gamma \quad \Gamma_0, \Gamma \vdash t : \forall \alpha : S. T}{\Gamma_0, \Gamma \vdash t[\alpha'] : [\alpha \mapsto \alpha'] T}$		
$\frac{\text{T-LETSYM} \quad \Gamma_0, \Gamma \vdash (\Lambda \alpha : S. t) : \forall \alpha : S. T \quad \alpha \text{ not free in } T}{\Gamma_0, \Gamma \vdash (\mathbf{let} \alpha : S \mathbf{in} t) : T}$	$\frac{\text{T-CTXABS} \quad \Gamma_0, \Gamma \cup \{\kappa\} \vdash t : T}{\Gamma_0, \Gamma \vdash \Lambda \kappa. t : \forall \kappa. T}$	$\frac{\text{T-CTXAPP} \quad \Gamma_0, \Gamma \vdash t : \forall \kappa. T \quad \not\exists (x : S) \in C}{\Gamma_0, \Gamma \vdash t[C] : [\kappa \mapsto C] T}$
$\frac{\text{T-QUOTE} \quad \Gamma, C \vdash t : T}{\Gamma_0, \Gamma \vdash [t] : \mathbf{Code} T C}$	$\frac{\text{T-ANTI0} \quad (x : \mathbf{Code} T \Gamma) \in \Gamma' \quad \not\exists (y : S) \in \Gamma}{\Gamma', \Gamma \vdash [x] : T}$	$\frac{\text{T-ANTI1} \quad (x : \mathbf{Int}) \in \Gamma'}{\Gamma', \Gamma \vdash [\mathbf{const} x] : \mathbf{Int}}$
$\frac{\text{T-ANTI2} \quad (\alpha : T) \in \Gamma'}{\Gamma', (\Gamma \cup \{\alpha\}) \vdash [\alpha] : T}$		
$\frac{\text{T-BINDSYM} \quad (\alpha : S) \in \Gamma' \quad \Gamma', (\Gamma \cup \{\alpha\}) \vdash t : T}{\Gamma', \Gamma \vdash (\lambda [\alpha]. t) : S \rightarrow T}$		
$\frac{\text{T-CLOSE} \quad \Gamma_0, \Gamma \vdash t : \mathbf{Code} T (C \cup \{\alpha\}) \quad \Gamma_0, \Gamma \vdash t' : \mathbf{Code} T C \quad (\alpha : S) \in \Gamma}{\Gamma_0, \Gamma \vdash \mathbf{close}_\alpha t \mathbf{else} t' : \mathbf{Code} T C}$		
$\frac{\text{T-RUN} \quad \Gamma_0, \Gamma \vdash t : \mathbf{Code} T \emptyset}{\Gamma_0, \Gamma \vdash \mathbf{run} t : T}$		

Value and context typing

$\frac{}{\emptyset \models \emptyset}$	$\frac{\Gamma \models \gamma \vdash v : T}{\Gamma \cup \{x : T\} \models \gamma \cup \{x \mapsto v\}}$	$\frac{\Gamma \models \gamma}{\Gamma \cup \{\alpha : T\} \models \gamma \cup \{\alpha \mapsto [x : T]\}}$
$\frac{\Gamma \models \gamma}{\Gamma \cup \{\kappa\} \models \gamma}$	$\frac{\text{T-CLOS} \quad \Gamma \models \gamma \quad \Gamma \cup \{x : T\} \vdash t : S}{\Gamma \vdash \langle \lambda x : T. t, \gamma \rangle : T \rightarrow S}$	$\frac{\text{T-KCLOS} \quad \Gamma \models \gamma \quad \Gamma \cup \{K\} \vdash t : T}{\Gamma \vdash \langle \Lambda K. t, \gamma \rangle : \forall K. T}$

Figure 7.2 – New typing rules for hygienic metaprogramming. The rules for match, rewrite and the subtyping rules are omitted, since they are similar to Figure 6.3.

Term evaluation

$\frac{\text{E-SCPAbs}}{\gamma \vdash \Lambda K. t \rightarrow \langle \Lambda K. t, \gamma \rangle}$	$\frac{\text{E-CTXAPP} \quad \gamma \vdash t \rightarrow \langle \Lambda \kappa. t', \gamma' \rangle \quad \gamma' \vdash [\kappa \mapsto C] t' \rightarrow v}{\gamma \vdash t[C] \rightarrow v}$
$\frac{\text{E-SYMAPP} \quad \gamma \vdash t \rightarrow \langle \Lambda \alpha : T. t', \gamma' \rangle \quad (\alpha' \mapsto v') \in \gamma \quad \gamma' \cup \{ \alpha' \mapsto v' \} \vdash [\alpha \mapsto \alpha'] t' \rightarrow v}{\gamma \vdash t[\alpha'] \rightarrow v}$	$\frac{\text{E-LET SYM} \quad \gamma \cup \{ \alpha \mapsto [x : S] \} \vdash t \rightarrow v \quad x \text{ fresh}}{\gamma \vdash (\text{let } \alpha : S \text{ in } t) \rightarrow v}$
$\frac{\text{E-CLOSED} \quad \gamma \vdash t \rightarrow [t'] \quad (\alpha \mapsto [x : T]) \in \gamma \quad x \notin FV(t')}{\gamma \vdash \text{close}_\alpha t \text{ else } t_e \rightarrow [t']}$	$\frac{\text{E-OPEN} \quad \gamma \vdash t \rightarrow [t'] \quad (\alpha \mapsto [x : T]) \in \gamma \quad x \in FV(t') \quad \gamma \vdash t_e \rightarrow v}{\gamma \vdash \text{close}_\alpha t \text{ else } t_e \rightarrow v}$

Quote evaluation

$\frac{\text{Q-ANTI2} \quad (\alpha \mapsto [x : T]) \in \gamma}{\gamma \vdash [\alpha] \rightarrow [x]}$	$\frac{\text{Q-BINDSYM} \quad (\alpha \mapsto [x : T]) \in \gamma \quad \gamma \vdash [t] \rightarrow [t']}{\gamma \vdash [\lambda \alpha]. t \rightarrow [\lambda x : T. t']}$
---	---

Extraction rules

$(t : T) \gg (t' : S)$	$= t \gg^T t' \quad \text{if } T <: S$	(X-ASC)
$x \gg^T x$	$= \emptyset$	(X-VAR)
$t \gg^T [x]$	$= \{ x \mapsto [t : T] \}$	(X-ANTI0)
$n \gg^T [\text{const } x]$	$= \{ x \mapsto n \}$	(X-ANTI1)
$x \gg^T [\alpha]$	$= \{ \alpha \mapsto [x : T] \}$	(X-ANTI2)
$(\lambda x : S. t) \gg^T (\lambda y : S'. t')$	$= [x \mapsto z] t \gg [y \mapsto z] t' \quad z \text{ fresh}$	(X-ABS)
$(\lambda x : S. t) \gg^T (\lambda [\alpha]. t)$	$= ([x \mapsto y] t \gg t') \cup \{ \alpha \mapsto [y : S] \} \quad y \text{ fresh}$	(X-BIND)
$(t_0 t_1) \gg^T (t'_0 t'_1)$	$= (t_0 \gg t'_0) \uplus (t_1 \gg t'_1)$	(X-APP)
<i>etc...</i>		

Figure 7.3 – Operational semantics of $\lambda^{[\alpha]}$. Only the new and updated rules are given; the rest are as in Figure 6.4.

body (ensured by T-ANTI0's new side condition); so while a pattern like $[\lambda x : \mathbf{Int}. x + 1]$ is still legal, a pattern like $[\lambda x : \mathbf{Int}. \lfloor y \rfloor + 1]$ is now illegal. In order to extract subterms from the bodies of bindings, one now has to also extract a corresponding variable symbol, as in: $[\lambda [\alpha]. \lfloor y \rfloor + 1]$, which brings $\alpha : \mathbf{Int}$ and $y : \mathbf{Code Int} \{ \alpha; \kappa \}$ into the typing context of the corresponding pattern-matching branch, assuming for example that the scrutinee has type $\mathbf{Code (Int} \rightarrow \mathbf{Int) } \{ \kappa \}$, for some unknown κ in scope.

Context substitution is defined as follows:

$$\begin{array}{ll}
 [\kappa \mapsto C] \mathbf{Int} & = \mathbf{Int} \\
 [\kappa \mapsto C] (\mathbf{Code } T D) & = \mathbf{Code } [\kappa \mapsto C] T [\kappa \mapsto C] D \\
 [\kappa \mapsto C] (S \rightarrow T) & = [\kappa \mapsto C] S \rightarrow [\kappa \mapsto C] T \\
 [\kappa \mapsto C] \Gamma & = \{ c \mid c \in \Gamma \wedge c \neq \kappa \} \cup \begin{cases} C & \text{if } \kappa \in \Gamma \\ \emptyset & \text{otherwise} \end{cases} \\
 [\kappa \mapsto C] x & = x \\
 [\kappa \mapsto C] (\lambda x : T. t) & = \lambda x : [\kappa \mapsto C] T. [\kappa \mapsto C] t \\
 [\kappa \mapsto C] (t_0 \ t_1) & = [\kappa \mapsto C] t_0 \ [\kappa \mapsto C] t_1 \\
 [\kappa \mapsto C] [\bar{t}] & = [[\kappa \mapsto C] \bar{t}] \\
 [\kappa \mapsto C] \Lambda \alpha : T. t & = \Lambda \alpha' : T. [\kappa \mapsto C] ([\alpha \mapsto \alpha'] t) & \alpha' \text{ fresh} \\
 [\kappa \mapsto C] \forall \alpha : T. T & = \forall \alpha' : T. [\kappa \mapsto C] ([\alpha \mapsto \alpha'] T) & \alpha' \text{ fresh} \\
 [\kappa \mapsto C] \Lambda \kappa. t & = \Lambda \kappa'. [\kappa \mapsto C] ([\alpha \mapsto \kappa'] t) & \kappa' \text{ fresh} \\
 \text{etc...} & \text{(other cases unsurprising)}
 \end{array}$$

Dynamic semantics

Figure 7.3 presents the main new parts of the dynamic semantics of $\lambda^{[\alpha]}$, compared to $\lambda^{\{\}}.$

An important question to answer is: how can we make sure that the usages of bindings extracted from patterns are indeed affine, when the term we extracted the binding from could be reused (and matched again) an arbitrary number of times? The answer is to refresh the binding we are currently matching before extracting it; this way, each extraction will yield an effectively distinct variable symbol. Notice how we use a fresh variable name y when extracting bindings in X-BIND.

Also notice that we do *not* have to rename the fresh variable z introduced in X-ABS in the result of the extraction (unlike in Figure 6.5), because we made sure that plain names were never extracted from patterns (by the side-condition of T-ANTI0).

7.4.3 Soundness

The soundness arguments for $\lambda^{[\alpha]}$ are mostly similar to those we have seen in Chapter 6 for λ^β . We define term annotation and evaluation $t \Downarrow v$ similarly, also implicitly adding all the adequate **err** evaluation rules.

One of the main differences is the addition of a context substitution lemma, which helps in proving progress and preservation:

Lemma 7.4.1 (Context substitution). *If $\Gamma_0, \Gamma \cup \{\kappa\} \vdash t : T$ then $\Gamma_0, \Gamma \vdash [\kappa \mapsto C]t : [\kappa \mapsto C]T$.*

Proof sketch. By induction on typing derivations. □

Theorem 7.4.2 (Preservation). *If $\vdash t : T$ and $t \Downarrow v$, then $\vdash v : T$.*

Proof sketch. Similar to theorem 6.3.11, by induction on evaluation derivations with a stronger lemma (like lemma 6.3.10). □

Theorem 7.4.3 (Progress). *If $\vdash t : T$ and $t \Downarrow v_e$, then $v_e \neq \mathbf{err}$.*

Proof sketch. Similar to theorem 7.4.3, by induction on typing derivations. □

Moreover, we can also state a *hygiene* theorem, which we phrase in terms of the absence of shadowing, since hygiene is about unintended variable capture, and unintended variable capture in a closed program normally manifests as the presence of shadowing:

Theorem 7.4.4 (Hygiene). *If $\vdash t : T$ and $\vdash t \rightarrow [t']$ then t' contains no variable shadowing.*

Proof. Left as an exercise for the reader. □

7.4.4 Straightforward Extensions

In this subsection, we present possible extensions to $\lambda^{[\alpha]}$ which are left as future work.

Lexical scoping

As mentioned before, lexically-scoped cross-quotation references using plain names are currently disallowed by $\lambda^{[\alpha]}$ but can be implemented in a straightforward way using first-class variable symbols, like it is done in Squid. An extension to $\lambda^{[\alpha]}$ could apply an elaboration phase to rewrite programs which make use of cross-quotation references into programs using fresh variable symbols, which can be done with a simple, local transformation.

Imperative Effects

To better mirror the capabilities of Squid, we could add imperative features to $\lambda^{[a]}$ such as mutable references. We expect this change to be straightforward and unproblematic; effects caused problems in work such as the original MetaML because the meaning of identifiers in program fragments was derived from the lexical scoping of the quotes — i.e., code values could not safely leave their scopes at runtime — and mutable references as well as exceptions could be used to violate this lexical scoping (pulling values out into the heap). However, various works have since shown [Kiselyov et al., 2016, Kameyama et al., 2015, Rhiger, 2012a] that properly reflecting scope dependencies inside the types of program fragments was sufficient to solve the problem.

Type-Parametric Matching

As seen in Sections 1.3.4 and again in 6.2.2, Squid has the ability to define patterns which extract *type representations*, in addition to subterms. Extending $\lambda^{[a]}$ with this functionality would require the extension of the type language to allow $[T]$ and $[T]$, and in order to prevent mixing up distinct extracted types we would need a mechanism to prevent extracted types from “escaping” the pattern matching branch in which they are available. The use of this extension would look as in the example below:

$$[(\lambda x : \mathbf{Int}. x + 1) 42] \text{ match } [(x : [T] \rightarrow \mathbf{Int}) y] \Rightarrow [x] [y] + [x] [y] \text{ else } [0]$$

Naturally, this feature would have to come with the GADT reasoning capabilities necessary to make type-parametric pattern matching on code values practically useful (see Section 1.3.12 and Appendix A).

7.5 Implementation in Squid

Here we quickly describe some aspects of the new scope-polymorphic system of $\lambda^{[a]}$ as implemented in Squid.

Hygiene and affinity

Since Squid is merely a macro library embedded in Scala, we have no easy way of enforcing the affine typing of variable symbols, which is required for full hygiene. Therefore, for now Squid users have to rely on discipline to stay out of trouble.

However, we have not found this to be a problem. The usual recommendation is to use cross-quotation references when possible (which hide the symbol handling as an implementation detail) and to keep bindings extracted from patterns local (as opposed to, e.g., leaking them into some wider scope and reusing them several times).

Interaction with plain names

In order to prevent the paradoxes of using plain names together with context polymorphism (see Section 7.1), Squid macros reject the handling of code which mixes plain names and abstract context, similarly to $\lambda^{[a]}$. For instance, Squid will reject the program `code" (x: Int) => $b"` if `b` has type `C { x: Int }`, which contains a plain name, but it will allow that expression if `b` has type `{ x: Int }`, which does not contain an abstract context component.

Integration with non-contextual metaprogramming

Maintaining precise context information throughout metaprograms can prove to be a hindrance, especially when the metaprograms do not perform very advanced or tricky transformations. We have found that some users of Squid prefer to avoid the use of contextual types, and would rather program against the simpler non-contextual metaprogramming interface introduced in Chapter 1.

Thankfully, it is easy to use *contextual* Squid in this *non-contextual* way: one simply has to use a type synonym `type Code[+T] = squid.Code[T, Nothing]`, where `Nothing` is the bottom of Scala's subtyping lattice, meaning that `Code[T]` refers to code values in *arbitrary* contexts which can never be satisfied. Naturally, one cannot run such a `Code[T]` value, but Squid provides a `tryClose` method on code values with the following signature:

```
type Code[+T, -C] {  
  def tryClose: Option[Code[T, {}]]  
  ...  
}
```

This method allows context-averse metaprogrammers to dynamically check that a `Code[T]` program fragment is closed, and to evaluate it as a result, as in:

```
myProgram // type: Code[Int]  
  .tryClose // type: Option[squid.Code[Int, {}]]  
  .getOrElse(throw new Exception("oops")) // type: squid.Code[Int, {}]  
  .run      // type: Int
```

We have found that first-class variable symbols *also* work well in non-contextual usages of Squid. Users can extract, insert, and match symbol usages even when they have chosen not to track contexts. Naturally, this is less hygienic than when contexts are tracked, since we cannot statically make sure that a matched variable occurrence really is supposed to be visible at the site where a pattern matches it. To nevertheless allow these uses, Squid has a special case where it allows matching *any* symbol occurrences when the context of the scrutinee is the bottom context `Nothing`.

7.6 Example Applications

We now see three application examples of programming with first-class variable symbols and context polymorphism in Squid.

7.6.1 Bindings reversal

To demonstrate the versatility of our approach, let us see how one would traverse a block of code, matching the variable bindings in that block, and then reverse the order of all the bindings which were traversed. Note that this is not always possible, since some earlier bindings might depend on later ones.

Assuming `reverseBindingsOrder` as the name of the function, and assuming that it returns an option (returning `None` if the transformation is not possible), example uses would be:

```
reverseBindingsOrder(
  code"val a = 1; val b = 2; val c = 3; a + b + c"
) == Some( code"val c = 3; val b = 2; val a = 1; a + b + c" )

reverseBindingsOrder(
  code"val a = 1; val b = a; val c = 3; a + b + c"
) == None
```

This is possible to express thanks to Scala's advanced path-dependent type system. In the metaprogram below, we again use first-class polymorphic functions (a feature of the upcoming Scala 3), to simplify the presentation:

```
def reverseBindingsOrder[T, C](p: Code[T, C]): Option[Code[T, C]] = {
  def go[T, C](p: Code[T, C], k: [A, B] => Code[A, B & C] => Code[A, B & C])
    : Option[Code[T, C]]
    = p match {
      case code"val $v: Int = $init; $body" =>
        go(body, [S, D] =>
          (cde: Code[S, D & C & v.C]) =>
            code"val $v = $init; ${ k[S, D & v.C](cde) }"
        ).flatMap(b => v.tryClose(b))
      case expr => Some(k(expr))
    }
  go(p, [A, B] => identity)
}
```

7.6.2 Encoding Cross-Stage Persistence for a Staged Database

Another interesting use case which demonstrates the flexibility of path-dependent metaprogramming with `Variable`, we consider the use case of encoding *substitution contexts* manually: the goal is to allow the manipulation of bundles called `Assignments`, which are made of: 1. a program fragment which may contain an arbitrary number of free variables, and 2. a mapping from each of these free variables to an actual *value* to be used in its place. We make these `Assignment` bundles useful by making them able to *runtime-compile* the program fragment they enclose, using their value mapping to provide values for each free variable.

We give the definition of `Assignment` and its helper function `assign` below:

```
trait Assignment[+Ctx] {
  self => // 'self' is used to refer to the current instance of Assignment

  // Abstract members:
  type Abs[T]
  def abs[T, C](pgrm: Code[T, C & Ctx]): Code[Abs[T], C]
  def applyAbs[T](f: Abs[T]): T

  // Concrete members:
  def compile[T](pgrm: Code[T, Ctx]): T = applyAbs(abs(pgrm).compile)
  def & [D] (that: Assignment[D]) = new Assignment[Ctx & D] {
    type Abs[T] = self.Abs[that.Abs[T]]
    def abs[T, E](pgrm: Code[T, Ctx & E]) = {
      val a = that.abs[T, Ctx & E](pgrm)
      self.abs[that.Abs[T], E](a)
    }
    def applyAbs[T](f: Abs[T]): T = that.applyAbs(self.applyAbs(f))
  }
}

def assign[Ty](v: Variable[Ty], x: Ty) = new Assignment[v.C] {
  type Abs[T] = Ty => T
  def abs[T, C](pgrm: Code[T, C & v.C]) = code"$v => $pgrm"
  def applyAbs[T](f: Abs[T]): T = f(x)
}
```

Each instance of `Assignment[Ctx]` is associated with an abstract type constructor `Abs[T]` which represents the type of arbitrary code `T` after having been wrapped inside the lambdas necessary to abstract away context `Ctx`. For example, given two variables `v0: Variable[Int]` and `v1: Variable[Double]`, for assignment `assign(v0, 0)` we get `Abs[T] = Int => T` and for composite assignment `assign(v0, 0) & assign(v1, 1.0)`, we get `Abs[T] = Int => Double => T`. Methods `abs` and `applyAbs` allow us to go back and forth between plain and abstracted forms, and method

compile uses these to compile and then evaluate a piece of code within a context where each variable of `Ctx` is associated with its assigned value.

Query compilers like the one we saw in Section 6.5 do not exist in a vacuum. At the time a new query is sent to a database system, the runtime data structures representing the various database tables may already be loaded in memory. In order for the compiled query program to access this data, we need a way to make the generated code, which lives in the *next stage*, refer to data that lives in the *current stage*.

For example, if the data of some `Person(Name, Age)` table is stored in an `ArrayBuffer[(String, Int)]` named `arr` at runtime, the code for a query that scans this table needs a way to refer to that `arr` so that when it is evaluated, it can access the stored data. In multi-stage programming, this is traditionally solved using *cross-stage persistence* (CSP). However, here we demonstrate that thanks to Squid's advanced capabilities to handle non-lexically-scoped open code and to abstract over variable symbols (see Section 7.4.1), we can avoid the use of CSP altogether. To do this, we make use of the `Assignment` data type described above.

In the following example, we define a `Table` data type that stores an `ArrayBuffer` of runtime values and provides a scan program fragment that performs an iteration on that data. Using this, we construct a program which prints the Cartesian product of two tables `t0` and `t1`, and then we compile/evaluate that program at runtime:

```
class Table(arr: ArrayBuffer[Int]) {
  val arrV = Variable[ArrayBuffer[Int]]
  val asnt = assign(arrV, arr)
  val scan = code"(k: Int => Unit) => $arrV.foreach(k)"
}

val t0 = new Table(ArrayBuffer(1, 2, 3))
val t1 = new Table(ArrayBuffer(4, 5, 6))

(t0.asnt & t1.asnt).compile(
  code"${t0.scan} { x0 => ${t1.scan} { x1 => println((x0, x1)) }}"
// ^ modulo runtime compilation and some normalization steps by the IR,
// the expression above is equivalent to:
t0.arr.foreach { x0 => t1.arr.foreach { x1 => println((x0, x1)) } }
```

The program above prints the tuples (1,4), (1,5), (1,6), (2,4), etc.

7.6.3 A safer take on flatMap streamlining

We now revisit the `flatMap` streamlining problem we saw in Section 5.7 in the context of stream fusion: we needed to analyze the argument in calls to the stream `flatMap` function, in order to separate the main lambda abstraction passed from its captured enclosing state, so that we

could make that state “resettable” by turning every bound value into a bound mutable variable that could be reset to its initial value at will. To achieve this, we had to resort in Figure 5.3 to using an unsafe scope extrusion mechanism that, when misused, could create unbound variable errors at runtime (the open function).

We now see a safer algorithm which achieves the same goal. This algorithm is a simplification of that of Figure 5.3 and does not handle as many cases.

The function below takes a program made of let bindings followed by one lambda abstraction, and turns that into a program which returns a tuple made of one lambda abstraction with the same semantics, along with an effectful thunk that, when executed, *resets* the state of that lambda. To simplify the handling of mutable variables, we assume the use a `MutRef` data type⁸ with the usual `!r` and `r := v` operations for getting and setting the value, respectively.

```
def rec[T, C](p: Code[T,C], reset: Code[Unit,C])
  : Option[Code[(T, () => Unit), C]]
= p match {
  case code"val $x: $xt = $xv; $body"
    => val v = Variable[MutRef[xt.T]]
      val newBody = x.substitute(body, code"!$v")
      rec(newBody, code"$reset; $v := $xv") match {
        case Some(r) =>
          Some(code"val $v = MutRef($xv); $r")
        case None => None
      }
  case code"($a: $ta) => $body"
    => Some(code"(($a => $body): T, () => $reset)")
  case _ => None
}
```

The `rec` function recursively analyses the program’s binding structure, wrapping each bound value into a `MutRef`, and accumulating a reset expression representing how to reset these references. Notice how we are able to recursively call `rec` on `newBody`, which has type `Code[T, C & v.C]`, thanks to our approach to hygienic context polymorphism.

As an example usage, consider the following invocation:

```
rec(
  code"val x = readInt; val y = MutRef(x); (a: Int) => {y := !y + 1; a + !y}",
  code"()"
)
→ Some(code" "
  val x_0 = MutRef(readInt)
```

⁸In fact, Squid transparently encodes mutable local variables as immutable local variables which use the `MutRef` data type.

```

val x_1 = MutRef(MutRef(!x_0))
(
  (a_1: Int) => {
    !x_1 := !(!x_1) + 1
    a_1 + !(!x_1)
  },
  () => {
    ()
    x_0 := readInt
    x_1 := MutRef(!x_0)
  }
)
" " ")

```

7.7 Related Work

We now review some related work.

Context polymorphism and scope safety

Context polymorphism (or scope polymorphism) in a contextual type system, also called *support polymorphism*, has already been investigated by several authors, such as e.g., Nanevski [2002], Kameyama et al. [2015], and Kim et al. [2006]. These systems have offered various different properties, but to the best of our knowledge none has proposed analytic metaprogramming with hygienic non-lexically-scoped open code manipulation.

If we ignore the analytic capabilities and non-lexical scoping, our system with first-class variables symbols and context polymorphism is very similar to the refined environment classifiers approach of Kiselyov et al. [2016].

Hygienic binding manipulation in metaprogramming

FreshML [Shinwell et al., 2003] is an extension of ML specifically designed to soundly manipulate variable bindings in metaprograms. FreshML has been an influential approach to dealing with bindings, in the context of user-defined abstract syntax trees. It uses first-class representations of names, which are passed around and can participate in the definition of functions like substitution, all the while making sure that hygiene is respected.

The first-class representation of names in FreshML has some similarities with our first-class representation of bindings; however, there is a major difference: in FreshML, the actual *values* of these representations do not appear in types, and thus types cannot express dependencies upon them. FreshML enforces hygiene via runtime mechanisms, but does nothing to prevent

scope extrusion. This is in contrast with our dependent system, which keeps track of specific variable representations in the type system, and statically prevents scope extrusion.

The Caml system by Pottier [2006] is an implementation of FreshML for OCaml. Pure FreshML [Pottier, 2007] is another version of FreshML which uses a “static discipline” for enforcing purity in FreshML metaprograms, meaning that name generation is not an observable effect. To achieve this, Pure FreshML uses a Hoare-style logic and an external (not part of the type system) fully-automated decision procedure. As presented, Pure FreshML does not support first-class functions or mutable state.

De Bruijn indices, as used by, e.g., Chen and Xi [2005] and many others, are an ad-hoc way to hard-code scope safety in the type system of a host language. Even though they can be used to enforce type safety, they are also patently unhygienic [Kiselyov et al., 2016].

Internal representation of bindings

Squid uses a scheme similar to the *locally named* representation [McKinna and Pollack, 1993]. This scheme consists in using different syntactic constructs to distinguish free variables from bound variables, so that they can never be confused. α equivalence between terms could be implemented more efficiently in Squid (it is currently done by checking mutual extraction) if we used either a *canonical* locally named representation [Pollack et al., 2012] or a *locally nameless* representation [McBride and McKinna, 2004, Charguéraud, 2012], but both have drawbacks in the context of metaprogramming and DSL compilers design — they forget the original names of variables, which are helpful when debugging.

Resource tracking in the type system

The problem of statically tracking the dependencies between different resources while preventing “leaks” can be found in many places in the programming languages literature. Tracking the presence of free variables in terms through the type system is just one specific instance of this more general problem.

In particular, there seems to be a lot of commonality between, on one hand, scope-safe hygienic metaprogramming, and on the other hand, techniques which have been used to enforce abstraction safety for algebraic effect handlers [Zhang and Myers, 2019, Biernacki et al., 2019].

A particularly close system to ours is the Scala Effekt library, by Brachthäuser et al. [2020], which coincidentally also uses contravariant type parameters holding intersections of abstract type members, in order to guarantee effect safety.

8 Multi-Stage Programming in the Large with Staged Classes

Despite its limitations (explored in Chapter 4), multi-stage programming (MSP) still holds great promises. Indeed, it allows generating specialized partially-evaluated code reliably, with the static guarantee that the generated code is well-typed and well-scoped. In principle, this readily gives to high-level languages the means to produce implementations which execute as fast as alternatives implemented in lower-level languages.

Yet, I argue that MSP has not reached its full potential yet, as it has been traditionally limited to generating *expressions*, and has lacked principled facilities for generating *modular programs* and *data structures*. In that sense, I argue that MSP has been reserved for programming “in the small,” focused on generating efficient kernels of computation on the scale of single function bodies, instead of complete applications.

In this chapter, I present a novel technique called *staged classes*, which extends MSP with the ability to manipulate class definitions as first-class constructs in a type-safe way. This lets programmers use MSP “in the large,” on the level of *applications*, rather than mere functions.

Using this technique, applications can be designed in an abstract and modular way without runtime cost in the steady-state, as staged classes guarantee the removal of all staging-time abstractions, resulting in the generation of efficient specialized modules and data structures.

In this chapter, we show that staged classes can be used for defining type- and scope-safe implementations of type providers. Our main application example, a prototype staged relational database system in Scala, is described in the next chapter.

8.1 Introduction

Multi-stage programming (MSP, or just *staging*) has been an intense subject of research in the past two decades [Parreaux et al., 2017c,a, Scherr and Chiba, 2015, Kim et al., 2006, Nanevski and Pfenning, 2005, Kameyama et al., 2015, Sheard et al., 1999, Taha and Nielsen, 2003, Kiselyov, 2017, Ofenbeck et al., 2013, DeVito et al., 2013, Carette et al., 2009, Taha and

Sheard, 1997, Taha, 1999, Ganz et al., 2001, Yallop and White, 2015, Rompf and Odersky, 2010, Rompf, 2016, Scherr and Chiba, 2014, Rompf et al., 2013, Taha and Sheard, 2000, Jonnalagedda and Stucki, 2015b, Oishi and Kameyama, 2017, Westbrook et al., 2010, Shaikhha et al., 2016]. This can be explained by the great promise that MSP offers: the ability to specialize or *partially evaluate* programs in a type-safe and modular way, using the abstraction capabilities provided by the general-purpose programming language itself.

This is in stark contrast with other common approaches to program specialization and generation, such as C++ templates. Indeed, templates rely on a relatively obscure and inflexible untyped metalanguage that makes their use for metaprogramming unnecessarily challenging.

Yet, despite their obvious drawbacks, C++ templates have enjoyed a colossal success in industry; they have come to underpin countless programs deployed in production, and to support the backbones of many performance-sensitive software infrastructures, such as the C++ Standard Template Library [Josuttis, 2012] and Boost [Abrahams and Gurtovoy, 2004]. Meanwhile, to the best of our knowledge, MSP has not seen much use outside of academia, with the occasional exception of very specific and narrow application domains such as computational kernels for heterogeneous systems [Sampson et al., 2017, Masliah et al., 2016, Haidl et al., 2016].

In this chapter, we argue that one limitation that has been holding MSP back from more widespread use is that it only tackles one part of the modularity problem. More specifically, it provides tools for modularizing the *program generation* side of a metaprogram, but does not provide tools for making the *generated program* itself modular. Indeed, traditional staging only deals with generating *expressions*, which makes it great for generating highly-efficient algorithmic kernels such as single database queries [Rompf and Amin, 2015b, Klonatos et al., 2014], single parsers [Jonnalagedda et al., 2014], single stream processing pipelines [Kiselyov et al., 2017, Jonnalagedda and Stucki, 2015b], individual linear algebra programs [Ofenbeck et al., 2013, Sujeeth et al., 2011], etc. However, it usually provides no dedicated facilities for generating data structure *declarations* and for sharing code across generated expressions and usually has limited code sharing capabilities across separate function bodies.

While this may seem like an insignificant limitation (one would be tempted to ask: “who cares about the modularity of the generated code?”), it turns out to have very concrete consequences on real-world applications. First, it is not always practical to generate entire programs into single expression. Generating reusable specialized components can be of central importance for keeping code sizes in check and making compilation times manageable. MSP leads to *always inlining* implementations, but inlining does not always improve runtime performance and can be counter-productive, so there should be mechanisms to control it. Second, in certain scenarios, such as the generation of specialized high-performance library code for humans to use, the modularity and reusability of the generated code is an important requirement.

So far, this limitation was typically tackled by pragmatic MSP approaches [Rompf and Odersky, 2010, Ofenbeck et al., 2013, Shaikhha et al., 2016] in ways that do not integrate well with

the rest of the MSP philosophy. Indeed, these approaches typically do not provide strong compile-time guarantees about the generated code, nor are they expressed in a high-level modular way. This is because the facilities provided by the MSP frameworks they rely on are essentially geared towards composing *expressions*, not definitions — so in order to generate modules, classes, and data structures, one often has to resort to ad-hoc string-based program generation.

To solve this shortcoming, we introduce *staged classes*, a new statically-typed abstraction that extends MSP by enabling the manipulation of object-oriented class definitions, while ensuring that the generated code is well-typed and well-scoped, in line with traditional MSP. We say that this approach enables *full modularity*, which we define as the property of a program generation system that allows *both* the program generation code *and* the generated code to be modular and shared. We argue that this “unleashes” the potential of MSP, making it a practical tool for structuring program generation in the large, at the scale of real-world applications. As a first step towards supporting this claim, we will detail the design of a fully-modular relational database management system prototype in Scala, using the Squid MSP system [Parreaux et al., 2017c] extended with staged classes.

We advocate two main usage modes for staged classes:

- As a way of defining generic libraries that do not pay for their genericity: individual users will specify the specializations that they are interested in, and will use the generated classes directly (for example, see the `Vector` class).
- As a way of taking MSP to the next level: we argue that allowing the generation of type and module declarations containing mutually-recursive methods is a big step forward that enables new usages of MSP “in the large,” as we will demonstrate with a real-world-inspired use case in the next chapter (Chapter 9).

To give an early intuition about staged classes, consider the following class (in Scala) which defines three mutable fields of type `Float`, all initialized to the value `0.0f`:

```
class Vector3 { var x0, x1, x2: Float = 0.0f }
```

We can represent that class as a *staged class* as follows. A staged class is an *instance* of class `squid.Class` (not to be confused with the standard `java.lang.Class` type), where fields are defined using a `varField` “virtualized construct” [Moors et al., 2012].

```
val Vector3 = new Class { val x0 ,x1 ,x2 = varField[Float](code"0.0f") }
```

Moreover, notice that we are now passing the initial field value inside a `code` quotation. This is because this is code destined to live in the “next stage,” i.e., in the resulting generated program. Given some term `t` of type `T` in context `C`, then `code" t "`, which has type `Code[T, C]`, is a first-class representation of term `t` that can be manipulated at run time.

The main advantage we gain from using staged class representations is the ability to programmatically abstract over different shapes of classes at no runtime cost. For example, consider the following generalization of `Vector3`, which is defined for arbitrary arity and component types:¹

```
class Vector[T](n: Int, init: Code[T,{}]) extends Class {  
  val xs = List.fill(n)(varField[T](init)) }
```

This is a staged class that abstracts over a family of related vector classes; the parameters `T`, `n` and `init` of this class are staging-time abstractions, and will disappear in the generated concrete classes. In particular, `new Vector[Float](3, code"0.0f")` represents a class that is equivalent to the original, unstaged `Vector3` class presented above.

8.2 Presentation of Staged Classes

In this section, we present staged classes and the Scala type system features they rely on. We do this through a progressive exploration of the design of a generic vectors library that abstracts over method implementations (Section 8.2.4), vector arity (Section 8.2.5), and element types (Section 8.2.6).

8.2.1 Classes in Scala

Consider the following Scala class, which represents a vector of three immutable floating-point coordinates:

```
class Vector3(val x0: Float, val x1: Float, val x2: Float) {  
  val sum = x0 + x1 + x2  
  def prod(v: Vector3): Float =  
    (x0 * v.x0) + (x1 * v.x1) + (x2 * v.x2)  
  def equals(v: Vector3): Boolean =  
    (x0 == v.x0) && (x1 == v.x1) && (x2 == v.x2)  
}
```

Several properties are hard-coded into this definition: the arity of the vector (it has three elements); the type of the elements; and also the runtime representation of instances of that class in memory — for example, the Java Virtual Machine currently dictates that an `Array[Vector3]` will be represented as an array of pointers to heap-allocated `Vector3` objects. None of these hard-wired properties can be generalized easily without losing either performance or modularity (or both). First, making a `Vector` class that is generic in arity or element type would incur copious amounts of boxing and indirection (due to the uniform representation principle of most high-level language runtimes). Second, leveraging more efficient memory representations, such as using three arrays of unboxed floating-point numbers to represent arrays of

¹ Function `List.fill(n)(v)` creates a list of `n` elements initialized to `v`.

vector instances (the so-called columnar representation), would force us to forgo the nicely encapsulated vector abstraction, making users have to deal with coordinates directly.

Moreover, consider the patent code duplication between the implementations of `prod` and `equals`. These two methods basically pair the elements of two instances using some operation (`*` and `==`, respectively) and then reduce the results with another operation (`+` and `&&`). Consider that in a real vector class, many other methods will share a similar structure, resulting in copious logic duplication. We could easily avoid such repetition by defining a `fold` operation, so that `prod` can be expressed as `fold(_ * _, _ + _)` and `equals` as `fold(_ == _, _ && _)`.² However, doing so would significantly degrade the overall performance, due to the use of higher-order functions, with no guarantees that the compiler would be able to remove these additional abstractions.

The result is that in practice, programmers who care about performance often have to resort to low-level and specialized implementations, which hampers modularity and incurs significant amounts of boilerplate and repeated code.

8.2.2 The Vector Class, Staged

The first step towards a solution to the problem laid out above is to represent our `Vector3` class as a staged class:

```
object Vector3 extends Class {
  val x0, x1, x2 = param[Float]
  val sum = field(code"$x0 + $x1 + $x2")
  val prod = method(
    code"(v:Self) => ($x0 * v.$x0) + ($x1 * v.$x1) + ($x2 * v.$x2)")
  val equals = method(
    code"(v: Self) => ($x0 == v.$x0) && ($x1 == v.$x1) && ($x2 == v.$x2)")
}
```

In Scala, `object` is used to define a singleton class instance. We instantiate the abstract class `Class`, which is used to represent “first-class” class representations. The `field` and `method` methods of `Class` are used to create members of the class being defined. Crucially, as the body of these methods, we use *program fragments*, which are quoted inside `code` quasiquotes. These represent code values that can be composed and inspected at run time. The dollar sign is used to escape quasiquotes and plug in some code fragments defined outside. In this example, we plug in the first-class field representations of the class. Finally, in the methods taking as a parameter an instance of the class being defined, we use the `Self` type, which refers to the current class. `Self` is an abstract type member (a type member whose definition is not exposed) defined in the `Class` supertype.

² ‘`(_ + _)`’ is shorthand for the lambda expression ‘`(x, y) => x + y`.’

Remark that our vector class is now an *object* (an instance of class `Class`), and the members of the class are now immutable *fields* of that object. The quoted `code` sections represent program fragments that will be composed together into the final generated program. Importantly, all code in black is code that only participates in the code generation process, and will *not* appear in the end program.

Generating the code. Finally, we can stringify the class represented by this `Class` instance by calling `Vector3.showCode`. The class thereby generated corresponds precisely to the `Vector3` class we showed at the beginning of this section. So, why go through the trouble of staging classes? The essential insight is that we are now free to generalize such class representations, and to introduce as much abstraction as we want on top of them: so long as these new abstractions are not part of the quoted program fragments (i.e., so long as they appear in black), they are guaranteed to be eliminated from the end program, so they will not incur runtime overhead.

Implicit naming of definitions. We claimed that the class generated for the staged `Vector3` object was identical to that of the original non-staged `Vector3` class. But how could that be, when we never specified the names of the class and of its fields, in our code? The trick is that the `Class` and `param` constructors (and many other staged class constructs) take implicit arguments, including a name argument that is resolved by default as the name of the enclosing definition.³ This way, a definition such as `'object C extends Class'` is in fact equivalent to `'object C extends Class(name = "C")'`, and `val f = field(v)` is equivalent to `val f = field(v)(name = "f")`.

Explicit Constructs vs Quasiquotation. The attentive reader will realize that the staged class infrastructure described here seems like a departure from the quotation-based approach we have so far favored in *Squid*. This naturally raises the question: *why not use quasiquotes for classes, too?*

One reason we prefer explicit constructs for staged classes is that staged class definitions typically contain lots of current-staged code and definitions *also* using the scope of the staged class for their own encapsulation. Moreover, we found that quotation did not lend itself well to field and method generation patterns based on side effects; as will become clear in the rest of this chapter, practical staged class applications make extensive use of such capabilities.

8.2.3 Staged Class Instantiation

While it is easy to instantiate a staged class *after* its code has been stringified and dumped into a source file, it is not as straightforward to instantiate a class at staging time, while the class is still represented as a run-time instance of `Class`. This is because we do not always

³ For this functionality, we use the macro-based `sourcecode` library, available at <https://github.com/lihaoyi/sourcecode>. An alternative is to modify the Scala compiler to support the feature natively, as done, for instance, by Scala Virtualized [Moors et al., 2012].

know statically the number and/or types of parameters (as with the `Vector` class of generalized arity in the next section). It would be unsafe to allow unchecked constructor calls, such as `code"new $Vector3(1,2)"` — in this particular case, there would be too few arguments for `Vector3`; we would generate `new Vector3(1,2)`, which would not type check.

The most straightforward way of making a staged class instantiable at staging time is to associate it with an “official” factory method. This is done by extending `FactoryClass[T]` instead of `Class`, where `T` is the type of the argument that the factory will accept. The only difference between `Class` and `FactoryClass` is that parameters in staged classes extending `FactoryClass` need to be given an initial value, which can be computed from the `factoryArg` code value available from `FactoryClass`, which represents the factory argument.

For example, here is how we could have associated a `Vector3` staged class with a factory taking as a parameter a `Float` value to initialize the vector’s fields:

```
object Vector3 extends FactoryClass[Float] {
  val x, y, z = param[Float](factoryArg) }
```

The code above will generate a method `apply` inside an object generated alongside the class having the same name as the class itself (in Scala, this is called a *companion* object; it replaces static members). `Vector3.showCode` now outputs:

```
class Vector3( val x: Float, val y: Float, val z: Float )
object Vector3{ def apply(arg: Float) = new Vector3(arg,arg,arg) }
```

A reference to this `apply` method is provided by `FactoryClass` as ‘`make`’, of type `Code[T => Self, {}]` (here, `Code[Float => Vector3.Self, {}]`). We can use it as follows:

```
val r: Code[Vector3.Self, {}] = code"${Vector3.make}(0.5)"

// which at run time evaluates to:
r == code"new Vector3(0.5, 0.5, 0.5)"
```

A common pattern for classes with a known, static set of fields is to use a tuple as the factory parameter. For example, when defining a staged `Person` class whose parameters `name` and `age` are known in advance, we can write (using methods `tuple._1` and `tuple._2` to access the tuple components):

```
object Person extends FactoryClass[(String, Int)] {
  val name = param(code"$factoryArg._1") // has type Param[String]
  val age  = param(code"$factoryArg._2") // has type Param[Int]
```

Finally, note that in this example, all intermediate tuples will be eliminated automatically by `Squid`, similarly to how `squid` eliminates β -redexes aggressively; for example:

```
Person(code"(Console.readLine, Console.readInt)")
```

```
// will evaluate to:
code"val arg = (Console.readLine, Console.readInt);
    new Person(arg._1, arg._2)"

// which will immediately reduce to:
code"new Person(Console.readLine, Console.readInt)"
```

8.2.4 Generative Programming to Avoid Repetition

For example, here is how we could define the generic fold function we alluded to earlier, in order to factor the common structure in the add and equals methods:

```
type F = Float; type B = Boolean // for conciseness
def fold[C](map: Code[(F,F) => F, C], red: Code[(F,F) => F, C]) =
  code"(v:Self) => $red($red($map($x0, v.$x0), $map($x1, v.$x1)), $map($x2,
    v.$x2))"
// This allows us to define:
val prod = method(
  fold(code"(_: F) * (_: F)", code"(_: F) + (_: F)"))
val equals = method(
  fold(code"(_: F) == (_: F)", code"(_: B) && (_: B)"))
```

Notice that fold is a current-stage method definition; as such, it will *not* be part of the generated program — we solely use it to generate the bodies of the prod and equals methods, which, after a step of β reduction (inlining function values, guaranteed by our staging framework), will be exactly the same as in the original definition. Here we have modularized our code, and yet we have incurred no extra overhead as the generated class is still precisely the same as Vector3.

8.2.5 Generalizing the Vector Arity

To generalize the arity of our staged vector class, we parameterize our Class instances with some staging-time integer representing the arity, and we programmatically generate a list of corresponding class parameters:

```
class Vector(n: Int) extends Class { // class, not object anymore
  // ...
```

We use List.tabulate(n)(f) to generate a list of n elements, defined in terms of their index idx. We pass explicit name arguments to mirror the naming convention of Vector3:⁴

⁴ Note that the framework will automatically rename any field whose name clashes with an existing field. As a consequence, if we had not provided an explicit 'name = "x" + idx' here, the fields would all implicitly receive the name xs of the nearest enclosing definition, but we would end up with actual field names xs, xs_1, xs_2, etc.


```
val xs = List.tabulate(n)(idx => param[Float](name = "x" + idx))
```

Next, we use the `reduce` function defined on Scala `List`s to create a program fragment that represents the sum of all elements of the current vector, and we assign the result to the usual `sum` field. For example, for `xs = List(x,y,z)`, we end up generating `val sum = field(code"$x+$y+$z")`.

```
val sum = field(
  xs.reduce[Code[Float,Ctx]]((lhs,rhs) => code"$lhs + $rhs")
```

Note that value `xs` has type `List[Param[Float]]`, but we want to reduce to a `Code` type; thankfully type `this.Param[Float]` is a subtype of `Code[Float, this.Ctx]` (where `Ctx` is the phantom type corresponding to the class' scope); we provided an explicit type argument to `reduce` to help type inference.

The generation of our element-fold implementation follows a similar logic, applying `map` on the `List` of elements first, and then `reduce`, using the provided functions `m` and `r`:

```
def fold[C](m: Code[(F,F) => F,C], r: Code[(F,F) => F,C]) =
  code"(v:Self) => ${
    xs.map(x => code"$m($x, v.$x)")
    .reduce((lhs,rhs) => code"$r($lhs, $rhs)") }"
```

Finally, the implementations of `prod` and `equals` remain unchanged. For illustration purposes, the class generated by calling `new Vector(2)(name = "V2").showCode` is:

```
class V2(val x0: Float, val x1: Float) {
  val sum = x0 + x1
  def prod (v: V2): Float = (x0 * v.x0) + (x1 * v.x1)
  def equals(v: V2): Boolean = (x0 == v.x0) && (x1 == v.x1)
}
```

8.2.6 Generalizing the Element Type

Preamble: Type Classes. As the last generalization step for our vector class, let us consider how to make a generic `Vector` that can accept different element types. First, the question is how to abstract over the addition and multiplication that are performed on `Float` values in our original class. The idiomatic way to do that in Scala is to use a type class [Oliveira et al., 2010]. We will use the `Numeric` type class, which is defined as:

```
class Numeric[N] {
  def plus(lhs: N, rhs: N): N
  def times(lhs: N, rhs: N): N
  ...
  /* and other methods */
}
```

```
}  
object Numeric {  
  implicit val ForFloat: Numeric[Float] = ...  
  ...  
}
```

The Scala implicit resolution mechanism, which happens during type checking, is responsible for finding appropriate instances of this class. Instances can be provided by users as well as by Scala's standard library in the `Numeric` companion object. If an implicit `Numeric[Float]` is required, Scala automatically synthesizes a reference to `Numeric.ForFloat`.

Type Class Lifting. In order to use this type class for type `N` in *quoted* code, we need an instance of `Code[Numeric[N], {}]`. Thankfully, Squid provides an implicit macro that automatically lifts implicit instances. For example, if an implicit `Code[Numeric[Float], {}]` is required, the Scala type checker will synthesize `code"Numeric.ForFloat"` for us.

Generic Vector. With all this out of the way, we generalize the element type of our `Vector` class as shown below, where `objectField` is used to create a field in the companion object:⁵

```
class Vector[N](n: Int)  
  (implicit ty: CodeType[N], numCode: Code[Numeric[N], {}])  
  extends Class {  
    val num = objectField(numCode)  
    val params = List.tabulate(n)(i => param[N](name = "x" + i))  
    val sum = method(  
      xs.reduce[Code[N, Ctx]]((lhs, rhs) => code"$num.plus($lhs, $rhs)")  
    )  
    def fold, equals ... // fold and equals are unchanged!  
    val prod = method(  
      fold(code"$num.times(_,_) ", code"$num.plus(_,_)")  
    )  
  }
```

The `ty: CodeType[N]` implicit value is required by Squid to manipulate code values of generic type `N`, as explained in Section 1.3.3.

Note: we did not have to change the definition of `equals` because it relies on the JVM `equals` method, which is available on all types (it's not based on a type class). Relying on JVM `equals` is idiomatic in Scala code — for instance, the collections of the standard library also do it.

Implicit Type Class Operations. In the code above, we have to call methods from the `Numeric` type class explicitly, as in `code"$numCode.plus(a, b)"` instead of `code"a + b"`. This is because there is no `Numeric[N]` implicit in scope to provide the usual `Numeric` syntax helpers — we only

⁵A Scala class may be accompanied by an object definition with the same name as the class, called the *companion object*, which can hold its own fields and methods. Companion object members serve the same role as static class members in more traditional object-oriented programming languages like Java and C#.

have a `Code[Numeric[N], {}]` in scope; a `Numeric[N]` implicit is needed to enable the shorthand syntax.

Thankfully, Squid provides a way to remedy this. Given a code value `cde` of type `Code[T, C]` or `Field[T]`, one can import `cde.unliftedImplicit`, which brings an implicit `T` in scope that can only be used within quasiquotes. This way, we can rewrite the `Vector` example using shorthand `Numeric` syntax, as in:

```
class Vector[N](n: Int) ... extends Class {
  ...
  import num.unliftedImplicit
  val sum = method(
    xs.reduce[Code[N, Ctx]]((lhs, rhs) => code"$lhs + $rhs")
  val prod = method(
    fold(code"_ * _", code"_ + _")
  ...
}
```

Generated Code. Instantiating the staged class above and inspecting the code it generates can be done by calling `new Vector[Int](3)(name = "IntVector3").showCode`, which produces code equivalent to:

```
object IntVector3 {
  val num = Num.ForInt
}
import IntVector3.num
class IntVector3(val x0: Int, val x1: Int, val x2: Int) {
  def sum = num.plus(num.plus(x0, x1), x2)
  // etc.
}
```

Since the element type is no longer generic, the element values are not boxed at runtime, which typically translates to important performance gains. Although `Numeric[N]` methods like `plus` are declared, in the `Numeric` interface, to take arguments of a generic type `N`, the actual `Num.ForInt` instance takes plain `Int` values (the Scala compiler generates “bridges” for such instances to still implement the generic `Numeric`’s methods); therefore, `num` method calls performed directly on `Num.ForInt` use plain `Int` values, without any boxing.

8.2.7 Direct and Staged Inheritance

Staged classes can extend existing *non-staged* classes or interfaces. This is done by mixing in [Odersky and Zenger, 2005] the `Implements[T]` trait (a trait defined by Squid for this purpose), where `T` is the type of all the interfaces to be implemented, for example: `object MyClass extends Class with Implements[MyInterface1 with MyInterface2 with ...]`.

The `Implements[T]` mixin refines the type of `Self`, making it a subtype of `T`, so that we can use the class accordingly. We use a macro to check at compile time that all the fields and methods required by the interfaces mentioned in `T` are provided with compatible types in the staged class. Naturally, dynamically-generated fields with names that are not constant string literals will not be accounted for in the check.

For future work, we are considering the possibility of letting staged classes inherit from other staged classes. However, we have not yet felt the need for such feature. This is likely because other ways of modularizing a staged class implementation are readily available and are sufficiently convenient, by the very nature of staged classes.

8.2.8 Staged Class Caching

In order to avoid code duplication, Squid provides a caching mechanism so that a staged class instantiated with the same staged parameters twice will only generate a single, shared class. To leverage this feature, users have to annotate their cached classes with the ‘cached’ macro annotation:

```
@cached class Example[A: CodeType](n: Int) extends Class { ... }
```

Where `[A: CodeType]` is a shorthand for `[A](implicit _: CodeType[A])`. The example above will generate, at compile time, a ‘cached’ method in the companion object:

```
// expansion of @cached macro:
class Example[A: CodeType](n: Int) extends Class { ... }
object Example {
  private val cache: Map[(Int, CodeType[_]), Example[_]] = Map.empty
  def cached[A](n: Int)(implicit _0: CodeType[A]) =
    cache.getOrElseUpdate((n, _0),
      new Example[A](n)(_0)).asInstanceOf[Example[A]]
}
```

Cached classes behave similarly to C++ templates, in terms of generated code modularity, as instantiations are shared.

8.2.9 Generic Methods

Staged classes offer two ways of defining generic methods:

- in a staged way, so that the method will be specialized (or *monomorphized*) at program-generation time — this is done by making the staged method itself a method; or
- by *generating* a generic method — the syntax for this alternative is with a prefixed type parameter list *in the body* of the method, as in ‘method(**code**“`[T] ...`”)’.

Importantly, the cached annotation described above also works with staged methods (the first kind of methods defined above), so that they also behave like C++ templates. For example, the following method:

```
@cached def foo[A:CodeType](n:Code[Int,{}]): Method[A=>List[A]] =
  method("(a: A) => List.fill($n)(a)")
```

generates a unique specialization for each unique pair of type A and value n that is passed to it.

8.2.10 Putting It All Together

Figure 8.1 shows the API of staged classes. It exposes some abstract types, such as `Field[T]`, which are types whose implementations are private to the staged class framework. These types can only be used through the staged class API, and through an extension made to Squid's quasiquotation system, which allows users to write expressions like `code"x.$f"` where x has type `cls.Self` and f is a field belonging to the staged class `cls`. The `Ctx` and `Self` abstract types are *phantom types*, in the sense that they are not used in the underlying implementation of staged classes, and merely serve to enforce static safety on the user's side. Notice how `Implements[T]` *refines* the type of `Self`, to mirror the fact that the class implements the interfaces in T, which can be used to type-check quasiquotes accordingly. The member-construction methods take by-name parameters (indicated with a leading `=>` as in `'body: => Code[T, Ctx]'`), which is used to delay their evaluation in order to allow mutually-recursive references to exist between methods, fields, and classes.

Since staged classes are built on top of the Squid framework [Parreaux et al., 2017c], we directly reuse Squid's machinery to achieve:

Type safety, which relies on Squid's type system extension.

Scope safety, via the `Ctx` type (remember that each `ClassLike` instance `cls` has a *distinct* type `cls.Ctx`, which prevents mixing up code fragments between different classes).

Cross-stage persistence for runtime compilation (see Chapter 9).

As a result, most type and scope errors in staged class metaprograms are caught at compile time, in the form of Squid errors or normal Scala type mismatches.

8.3 Use Case: Typed Type Providers

Sometimes, programmers want to avoid having to write repetitive and error-prone type definitions; yet, they are not necessarily willing to pay for the run-time overhead of approaches that abstract over runtime type representations, such as dynamic dictionaries. Languages like F# have provided facilities for generating types programmatically, via a popular feature known as *type providers* [Syme et al.], which has notably been used to provide types for external data

```
class ClassLike {
  type Ctx
  type Self
  type Method [T]
  type Field [T]
  type Param [T]

  def method [T: CodeType] (body: => Code[T, Ctx])
    (implicit name: Name) : Method[T]
  def field [T: CodeType] (value: => Code[T, Ctx])
    (implicit name: Name) : Field[T]
  def varField [T: CodeType] (value: => Code[T, Ctx])
    (implicit name: Name) : Field[MutRef[T]]
}
class Class(implicit name: Name)
extends ClassLike {
  def param[T: CodeType](implicit name: Name): Param[T]
}
class FactoryClass[T](implicit name: Name, ty: CodeType[T])
extends ClassLike {
  def param[T: CodeType] (init: Code[T, Ctx])
    (implicit name: Name) : Param[T]
  val factoryArg: Code[T, Ctx]
  val make: Code[T => Self, {}]
}
trait Implements[+T] {
  type Self <: T
}
```

Figure 8.1 – The core API of staged classes.

sources automatically [Petricek et al., 2016]. Type providers typically provide not only bare data types, but also utilities like serialization and deserialization to and from external data formats.

Problem. The main problem we identify with type providers is that they are not statically typed metaprograms: it is possible to define type provider implementations which mismanage the types and scopes of the code they generate. Such errors in the implementation of type providers may result in generated code that does not compile, or compiles with the wrong semantics (due to name hygiene problems). Importantly, such errors may only be detected when *users* of the type providers invoke them, and not necessarily when the type provider is designed by the metaprogrammer, so this can result in subpar user experience.

Solution with staged classes. In this section, we argue that staged classes can solve the problem of statically-typed type providers elegantly and concisely, relying on the well-known and proven principles of multi-staged programming.

8.3.1 An Embedded DSL for Record Type Providers

Consider the core problem of programmatically generating record types given sequences of field name/field type associations. We would also like to generate, along with these types, some automatic deserialization code to construct these records from, say, plain CSV files. Here, we show how to implement this in a type-safe way with staged classes.

We will settle on the following user-facing domain-specific language (DSL) for expressing record types, which is *embedded* [Hudak, 1996] in Scala's type system. In this example, we define a type `Person` which should have fields `name` of type `String` and `age` of type `Int`:

```
val Person = recordOf.name[String].age[Int].toClass
```

To make this work, we introduce a type class `Serial[T]` with method `scan(sc: Scanner): T`, used to efficiently retrieve a value of type `T` from some existing `java.util.Scanner`. We will provide default `Serial` implementation for common types, such as `Serial.ForInt: Serial[Int]`.

The `Person` definition above should produce the following class and factory method:

```
class Person(val name: String, val age: Int)
object Person {
  def apply(sc: Scanner): Person = {
    val name = Serial.ForString.scan(sc)
    val age = Serial.ForInt.scan(sc)
    new Person(name, age)
  }
}
```

8.3.2 Implementing the Type Provider DSL

To implement our record DSL, we use the builder pattern.⁶ We expose the following interface, where `Dynamic` is a standard Scala trait which we explain later in this section:

```
type RecBuilder <: Dynamic {  
  def toClass(implicit clsName: Name): FactoryClass[Scanner]  
  def selectDynamic[A](name: String)  
    (implicit srl: Code[Serial[A], {}], tpe: CodeType[A]): RecBuilder  
}  
val recordOf: RecBuilder
```

The `recordOf` entry point is a value of type `RecBuilder`. Type `RecBuilder` has a `toClass` method to generate the staged class corresponding to the record type built so far. Notice that `toClass` takes an implicit parameter of type `Name`, which will pick up the name of the nearest enclosing definition, unless specified explicitly by the user (as explained in Section 8.2.2).

Adding a field to the record builder is the most complex part of the API. We use a method called `selectDynamic`, which takes four parameters, the latter two being implicit:

- the type `A` of the field to be added;
- the name of the field to be added;
- an implicit code representation of a `Serial[A]` type class instance, which will be used to generate references to the appropriate scan methods; and finally
- an implicit type representation of type `CodeType[A]`, which *reifies* the type parameter `A`, and is required by `Squid` to manipulate program fragments in a generic context.

The last missing ingredient to making our DSL work is a little bit of magic that the Scala compiler can do for us: since `RecBuilder` is marked as extending `Dynamic`, the type checker automatically interprets calls of the form `rec.field[T]` as meaning `rec.selectDynamic[T]("field")`. This way, for example `'val P = recordOf.name[String].age[Int].toClass'` is equivalent to:

```
val P = recordOf.selectDynamic[String]("name").selectDynamic[Int]("age").toClass
```

Which, after typing and implicit resolution, is equivalent to:

```
val P =  
  recordOf  
    .selectDynamic[String]("name")(  
      squid.codeTypeOf[String], code"Serial.ForString")  
    .selectDynamic[Int]("age")(squid.codeTypeOf[Int], code"Serial.ForInt")  
    .toClass(Name("P"))
```

⁶The idea of the builder pattern is to compose an object by progressively specifying its properties, one by one, which is done by calling various methods on it.

Where `codeTypeOf` is a Squid macro that synthesizes the runtime representation of a Scala type.

The full, type-safe implementation of the record-building DSL is given below.

```
package DSL

abstract class RecBuilder extends Dynamic {
  private[DSL] def mkParams(cls: FactoryClass[Scanner]): List[cls.Param[_]]

  def toClass(implicit clsName: Name)
    = new FactoryClass[Scanner] {
      val params = mkParams(this) }

  def selectDynamic[A: CodeType] (name: String)
    (implicit srl: ClosedCode[Serial[A]])
    = new RecBuilder {
      def mkParams(cls: FactoryClass[Scanner]) =
        cls.param(code"$srl.scan({cls.factoryArg})")(name) ::
          RecBuilder.this.mkParams(cls) }
}

val recordOf = new RecBuilder {
  def mkParams(cls: FactoryClass[Scanner]) = Nil
}
```

Let us take some time to unpack this code, which is rather complex, and understand how it works. We use an internal `mkParams` method, which is private to our DSL package, in order to reconstruct a list of the class parameters to build once `toClass` is called. This has type `List[cls.Param[_]]`, because it is a list of parameters of the class `cls`, of unspecified type (hence the “wildcard” underscore in `Param[_]`). The list is built up inductively, starting from `Nil` in the base `recordOf` case, and adding elements using the `::` infix list-building operation.

For illustration purposes, consider that if we were to inline all the calls to `selectDynamic` and `toClass` that arose in the `P` example defined above, we would obtain:

```
val P = new FactoryClass[Scanner] {
  val fields = param[String](code"Serial.ForString.scan($factoryArg)")("name")
    :: param[Int](code"Serial.ForInt.scan($factoryArg)")("age")
    :: Nil
}
```

8.3.3 Type Provision From Data Samples

Following Petricek et al. [2016], we now would like users being able to have the type providers themselves infer the record names and types from an existing data sample:

```
val PersonModule = csvProvider("Person", "people.csv")
import PersonModule.Person
val p = new Person("Bob", 42) // using the explicit constructor
val q = Person(new Scanner(new File("data.csv"))) // using the CSV-record factory
```

And have the correct type inferred from the CSV sample in “people.csv”. This can be done easily by adapting the F# Data algorithm presented by Petricek et al. [2016]. Below, we present a simplistic implementation⁷ that only looks at the header and first data line of the CSV, and only tries to deserialize integers, booleans and strings. Our goal is to showcase the way staged classes allow for concise yet type-safe implementations of F# Data-style type providers; it is easy to imagine how to generalize such an implementation to handle more cases.

```
def inferShape(csvContent: Scanner): RecBuilder = {
  val headerNames, firstLineValues = csvContent.nextLine().split(',')
  var rec = recordOf
  for ((headerName, value) <- headerNames.zip(firstLineValues)) {
    if (hasIntFormat(value))
      rec = rec.selectDynamic[Int](headerName)
    else rec = rec.selectDynamic[String](headerName)
  }
  rec
}
```

This method parses the first two lines of a CSV file content (the *header names* and *first-line values*), pairs them up accordingly (using the `zip` method in `List`), infers the type of the field based on the `hasXFormat` functions, and updates a record builder to add each corresponding field name/field type.

As an example, the following definition of the `Person` class is equivalent to the one we defined in the previous subsection using the explicit record builder DSL:

```
val Person = inferShape(new Scanner("name,age\nBob,42")).toClass
```

As promised, we can now define a `csvProvider` method accepting a file name as an input, and which will generate a class at compile time. This can be done using a Scala macro [Burmako, 2013]. We omit the macro definition here, for lack of space.

⁷In Scala, `val a, b = E` is equivalent to `val a = E; val b = E`.

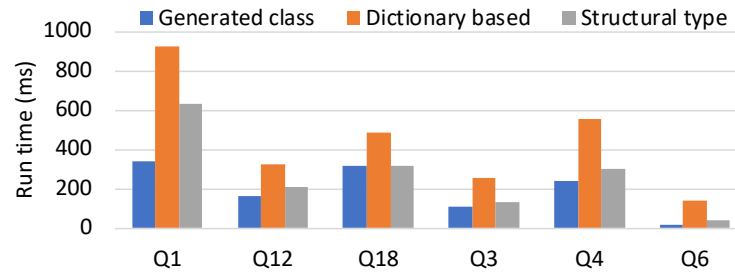


Figure 8.2 – Performance of different record representations, for computing a set of TPC queries.

8.3.4 Evaluation

Type Safety

By virtue of the contextual staged code types (see Chapter 7), the generated code is guaranteed to be well-scoped and well-typed. Note that this is true even if field names collide: in this case, the generated class will rename the duplicated fields following a predictable scheme based on the order of definition of the fields; for example, given three fields which are all assigned the same name “foo,” the generated class will contain fields named `foo`, `foo_1`, and `foo_2`.

Moreover, it is easy to see that if the data sample used in `inferShape` is representative, we will not generate programs that will go wrong at run time (see [Petricek et al., 2016] for more details).

Performance of Generated Classes

We now provide an answer to the question: “*Is all this complication worth it? Why not use a simpler, more dynamic approach to data records?*” Indeed, there are different approaches for representing records that do provide static type safety, but do not rely on program generation to achieve it. These approaches are usually based on dynamic dictionaries or on runtime reflection “behind the scenes.” In our opinion, the argument is then mostly about performance. In Figure 8.2, we show the performance difference between three different type-safe record representations in Scala, for a set of standard TPC queries [Transaction Processing Performance Council, 1999]: we benchmark plain classes as generated by our staged classes; dictionary-based records;⁸ and Scala structural types, which rely on anonymous classes and cached Java run-time reflection. We can see from the results that generated classes are consistently faster than both approaches: from 30% to an order of magnitude faster than dictionaries, and from about the same speed for Q18 to twice faster than structural types.

⁸ Provided by the Scala Records macro-based library, available at <https://github.com/scala-records/scala-records>.

8.4 Related Work

	CGM	GCM	DYN	STA	TYP
Staged Classes (This Chapter)	●	●	●	●	●
Squid (MSP) [Parreaux et al., 2017c]	●	○	●	●	●
MetaML (MSP) [Taha and Sheard, 2000]	●	○	●	○	●
LMS (MSP) [Rompf and Odersky, 2010]	●	○	○	●	●
MacroML (MSP) [Ganz et al., 2001]	●	○	○	●	●
Modular macros [Yallop and White, 2015]	●	○	○	●	●
AnyDSL/Impala [Leißa et al., 2015]	●	◐	○	●	●
Template Haskell [Sheard and Jones, 2002]	●	◐	○	●	◐
Scala “def” macros [Burmako, 2013]	●	○	○	●	○
Scala annot. macros	●	●	○	●	○
Scala @specialized	○	●	○	●	●
Scala miniboxing [Ureche et al., 2013]	○	●	○	●	●
Scala ADTR [Ureche et al., 2015]	◐	●	○	●	●
Exotypes [DeVito et al., 2014]	●	●	●	○	○
Type providers [Syme et al.]	●	●	○	●	○
.NET generics [Kennedy and Syme, 2001]	○	●	◐	◐	●
C++/D templates [Veldhuizen, 1998]	◐	●	○	●	○
D mixins ⁹	●	●	○	●	○
Rust traits	○	●	○	●	●

Table 8.1 – Comparison of existing approaches to specialization and partial evaluation via program generation. We assess whether the system offers: • modularity in the *code generator* (CGM), which we also view as the ability to programmatically configure the code generation process; • modularity in the *generated code* (GCM); • run-time specialization, from runtime values (DYN); • compile-time specialization (STA); • type safety: the static guarantee, at *compile time of the code generator*, that the generated code will be well-typed (TYP).

In this section, we review the related work. Table 8.1 summarizes existing approaches and their supported features.

Program generation is ubiquitous in software engineering, and can be used for both reducing boilerplate (to express some repetitive, verbose, or sometimes error-prone idiom) and improving performance (by removing the abstractions that get in the way of performance). C++ templates [Vandevoorde and Josuttis, 2002], Lisp and Rust macros [Kohlbecker et al., 1986], and Template Haskell [Sheard and Jones, 2002] are examples of program generators. An important factor to consider, while comparing the different alternatives, is that program generation (and metaprogramming in general) is hard and error-prone, so it is valuable to use systems that offer strong static type safety guarantees.

Multi-stage programming (MSP) is a technique for specializing programs in a type-safe and modular way, using the abstraction capabilities provided by the targeted programming lan-

guage itself. There are many MSP frameworks developed in different programming languages such as ML-like languages [Taha and Nielsen, 2003, Taha and Sheard, 1997, Taha, 1999], Scala [Rompf and Odersky, 2010, Rompf et al., 2013, Parreaux et al., 2017c,a], and even Java [Westbrook et al., 2010]. All these program generation and MSP frameworks only deal with generating expressions, and typically do not provide facilities for generating data structure declarations or for sharing code across generated expressions, although some forms of sharing have been devised in various works, as described in the next paragraph.

Swadi et al. [2006] generate shared code for staged memoized functions, with applications to dynamic programming. *Let insertion* is a common technique to generate shared definitions during partial evaluation and staging (see, for instance, [Kameyama et al., 2015]), and MetaOCaml introduced a primitive called *genlet* to facilitate it [Kiselyov, 2018], which can also be used to generate mutually-recursive definitions [Yallop and Kiselyov, 2019]. Staged classes offer yet another way of generating mutually-recursive definitions, where the locus of the generated definitions is fixed (it is the scope of the class) similar to one of the alternatives proposed by Yallop and Kiselyov [2019] except that staged classes also ensure scope safety — it is not up to the user to make sure the definitions are inserted in the valid scopes, as opposed to the solution proposed by Yallop and Kiselyov. As an interesting aside, I independently implemented a technique very close to the *genlet* approach of MetaOCaml (which automatically inserts the bindings in the widest appropriate scope) as part of one of my different ANF Squid intermediate representation prototypes, while trying to solve the same problem.

Another recent line of work has started to look into staging at the level of modules. Inoue et al. [2016] proposed using records and polymorphism to emulate the staging of ML modules. However, such manual encodings become cumbersome fast, especially when mutual reference and recursion are desired in the generated code, and when several generated modules or data structures need to inter-operate in a well-typed manner, as in our staged database example (see Chapter 9). Watanabe and Kameyama [2017] proposed and implemented a language called $\lambda^{<M>}$ which supports staged modules by translation into plain MetaOCaml, but their translation could lead to exponential code duplication. Sato et al. [2020] later corrected the problem by leveraging MetaOCaml’s *genlet* mechanism (mentioned in the previous paragraph), which allows performing let insertion automatically to avoid code duplication. A typical worry in the context of staging modules is that of extruding local module references — taking them out of their scopes, where they lose their meaning; we do not have this problem with staged classes, because all the generated classes are top-level definitions.

Terra [DeVito et al., 2013] is an MSP extension to the Lua programming language which uses MSP and Exotypes [DeVito et al., 2014] in order to generate type declarations. However, due to the dynamic nature of Terra, Exotypes do not benefit from the static safeties provided by our system. Metaphor [Neverov and Roe, 2004] is a multi-stage language with a type system that allows statically checking reflection-based field accesses. Ur [Chlipala, 2010] uses dependent types to allow first-class computation of records and names. Both of these systems provide static safety guarantees; however, they do not manipulate or create class definitions

programmatically.

ADTR [Ureche et al., 2015] is a Scala compiler plugin enabling programmers to modify the data representations in delimited scopes. This framework subsumes other frameworks such as Scala miniboxing [Ureche et al., 2013], however, is not as powerful as MSP frameworks in terms of code generation facilities.

F# type providers [Syme et al., Petricek et al., 2016] allow inferring types and methods based on external data. Type providers are fundamentally not type-safe. Hence, such errors may only be detected during their usage, not when they are designed by the metaprogrammer. Google protocol buffers is a domain-specific framework for generating serializable types [Google, 2008].

8.5 Conclusion

In this chapter, we have shown how to take multi-stage programming further than its traditional area of application (metaprogramming in the small), to fill a niche that was previously not occupied satisfactorily by any other approach (type-safe metaprogramming in the large), to the best of our knowledge. We have shown how to use staged classes to define generic and fully-modular libraries that do not pay for their genericity and modularity. We have demonstrated in detail two main use cases: a statically-typed type providers implementation, and a dynamic relational database compiler prototype. We explained the limitations of previous approaches, and how staged classes could allow for the design of more modular, practical, and reliable metaprogramming systems in the future.

9 Application: A Staged Database Compiler

In this chapter, we point out the deficiencies of previous staged query compilation approaches and motivate the use of staged classes to solve these deficiencies.

We describe the design of a modular *database compilation* system, which goes further than pure query compilation in that it generates efficient shared code for the database infrastructure itself, as well as for individual queries.

Our database system prototype is particularly interesting in that it makes unique use of advanced multi-stage programming (with more than two stages) and cross-stage persistence [Taha and Sheard, 2000]. Though it is very much a “toy” system compared to existing industrial database systems, we believe that it is easy to imagine how to enrich and extend it, as its design is fundamentally modular.

We start by assuming that our database system receives plain-text SQL queries as input, which may be ill-typed, and thus require some fallible dynamic checking; then, we see how to use Squid to embed a type-safe Scala DSL as an alternative frontend to our database system. The Scala EDSL has the advantage of being safer and more expressive than SQL, reducing the impedance mismatch between the application and the database.

We show how the Scala query EDSL is transformed into internalized query representations on which to perform rewritings, and then into query plans from which we can synthesize efficient implementations. We do this while ensuring type- and scope-safety statically, thanks to Squid’s advanced Scala-powered type system.

9.1 Motivation

In the era of “big data,” the amount of information to process and analyze is becoming a bottleneck for many data-oriented tasks. On the other hand, random-access memory is cheaper than ever, which explains the advent of in-memory databases. In this context, where cold disk memory latencies no longer dominate data analytic run times, economizing CPU

cycles has become crucial, and as such query compilation has been regaining interest in research [Klonatos et al., 2014, Shaikhha et al., 2016, Rompf and Amin, 2015b, Tahboub et al., 2018] and industry [Armbrust et al., 2015].

Approaches based on multi-staged programming [Klonatos et al., 2014, Rompf and Amin, 2015b, Shaikhha et al., 2016, Parreaux et al., 2017c, Cheney et al., 2013, Tahboub et al., 2018] have invariably focused on compiling individual queries into efficient but non-modular computation kernels. Typically, specialized data structures are generated as part of such kernels in an ad-hoc way (generally by cobbling strings together). There are two problems with that approach:

- First, the generation of these data structures is brittle and unsafe as it does not rely on any principled abstraction. This contrasts starkly with the rest of these systems, which is based on type-safe MSP, and thus benefits from all the reliability guarantees offered by MSP. Therefore, we argue that the data structure generation part of these systems constitutes a weak link in staged compilation pipeline.
- Second, in these approaches the entire database infrastructure usually has to be generated from scratch *for each individual query*. Workarounds to share this infrastructure between queries are certainly conceivable, but are doomed to be even more brittle, due to the untyped nature of these ad-hoc data structure generation approaches (see the first bullet).

In this section, we argue that staged classes are a perfect fit for the reliable generation, at run time *or* at compile time, of the backbone infrastructure of data structures underlying the implementation of database systems. This way, we propose to go *beyond* pure staged query compilation, introducing “staged database” compilation.

9.2 Architecture of the Staged Database System

In this chapter, we describe a (much simplified version of a) database system that is as close to real-world use cases as possible. As such, we focus on an *online* in-memory SQL-based service, which receives requests for data definition (table creation, deletion) and manipulation (loading, dumping), as well as SQL-style queries that are cached by default so that their compiled code can be reused later (we only examine a trivial SQL query, sufficient to understand the contribution of staged classes in this context, but we see more advanced queries later on in the chapter). These database requests will trigger the generation of specialized code to perform the underlying operations in an efficient manner.

9.2.1 Specialized Container Classes

Managing data storage efficiently is one of the foremost concerns in the implementation of a database system. The first thing we focus on is to define a set of containers for storing data tables in memory without overhead. This is not straightforward, first because often in object-oriented languages such as Scala, primitive values are boxed when they appear in generic contexts, and second because objects are allocated on the heap, which we seek to avoid. Here, we define a `Container[A]` staged data structure based on a dynamic array, which is specialized for each element type `A`, so that it does not box. The `[A: CodeType]` syntax is a shorthand for defining an implicit parameter in Scala, meaning `[A](implicit _: CodeType[A])`.

```
@cached class Container[A: CodeType] extends FactoryClass[Unit] {
  val initArraySize = code"16"
  val size = varField(code"0")
  val array = varField(code"new Array[A]($initArraySize)")
  val at = method(code"(idx: Int) => $array(idx)")
  val add = method(code"(a: A) => {
    if ($size == $array.length) {
      val old = $array
      $array = new Array[A]($size * 2)
      Array.copy(old, 0, $array, 0, $size)
    }
    val idx = $size; $array(idx) = a; $size = idx + 1;
    idx
  }")
}
```

As a side note, remark how `initArraySize` is defined as a separate code value; this allows users of the data structure to later override that value, and pick a possibly-non-constant value for it instead.

Note that explicitly-specialized containers are not actually needed in environments with support for run-time generic specialization, like C# and its .NET runtime system. However, such automatic specialization is *not* sufficient for achieving our performance goals, as we see in the next section, where the additional power of staged classes is required.

9.2.2 Column Store Meta-Container Class

It is not enough to specialize our dynamic array containers. We also want to avoid allocating record objects on the heap. The column-oriented format is the most appropriate for our use case, as it not only avoids allocating object wrappers, but has also been known by the database community to enable faster data access for common queries. We believe that staged classes are one of the first approaches to allow the type-safe yet modular and convenient definition of column-store containers, that do not pay for their modularity.

Chapter 9. Application: A Staged Database Compiler

Our `ColumnStore` container class is *parameterized* with another class representing the record type that we would like to store. Importantly, note that we will only use this class type as a module containing field information, and we will never actually create instances of it! This ensures that we will never pay for allocating record objects at run time.

```
class ColumnStore[A: CodeType](elemCls: FactoryClass[A])
  extends FactoryClass[Unit] {
  // ...
```

The most interesting aspect of the `ColumnStore` class is that we will need to store a collection of columns, where each column represents a parameter in the element class and is associated with its own specialized container class *and* with a `cntr` field of corresponding type in the `ColumnStore` class. We achieve this using a nested class `Column[T]`, that packages all this information together:

```
class Column[T](val field: elemCls.Param[T], val cntrCls: Container[T]) {
  val cntr: Param[cntrCls.Self] = param(code"${cntrCls.make}()")
}

val cols: List[Column[_]] = elemCls.parameters.map {
  case p: elemCls.Param[t] => new Column[t](p, Container.cached[t]) }
```

We can now define the `add` method, which builds a sequence of mutating statements in a local variable `res`. The method works by leveraging the `initValue` available in each `Param[T]` object of `FactoryClass[A]` instances, which has type `Code[A => T, {}]`. For each column, we build the corresponding parameter value, and then add it to the corresponding specialized container, returning the index as expected:

```
val firstColumn = c(0)
val size = method(code"() => ${firstColumn.cntr}.${firstColumn.cntrCls.size}")
val add: Method[A => Int] = method(code"(arg: A) => {
  var res: Code[Unit, arg.type] = code"()"
  for (c <- cols) {
    res = code"$res; ${c.cntr}.${c.cntrCls.add}(${c.field.initValue}(arg))"
  }
  code"$res; $size() - 1"
})")
```

Finally, we need to provide a `getFieldAt` method, which will allow users to retrieve a particular field at a particular index in the columnar container. To do this, we find the corresponding column field in our list of columns, and use the `at` method of its container. Note that this is a staged method with one static part (the field parameter) which will disappear from the generated code, and one dynamic part (the index parameter).

```
@cached def getFieldAt[T](f: elemCls.Param[T]): Method[Int => T] = {
```

```

    val c = cols.find(_.field == f).get
    method(code"(idx: Int) => ${c.cntr}.${c.cntrCls.at}(idx)")
  }
} // end of ColumnStore

```

9.2.3 Loading and Emitting Data Efficiently

Let us now see how to handle our first database requests, related to data definition and manipulation. When a user sends a CREATE TABLE request, we'll be provided with a request object of the form:

```

class TableCreationRequest
    (val tblName: String, val columns: List[(String,String)])

```

First, we want to generate a class with the following interface (**trait** is like Java's interface), which will be able to perform efficient loading and dumping of data according to the provided schema:

```

trait Table {
    def loadFrom(in: InputStream): Unit
    def dumpInto(out: OutputStream): Unit
}

```

We will reuse the recordOf/selectDynamic/toClass infrastructure of Section 8.3 to facilitate our job:

```

class ColStoreTable(r: TableCreationRequest)
    extends FactoryClass[Unit](name = r.name + "_Table") with Implements[Table]{
    val recordCls = columns.foldLeft(recordOf){
        case (row, (colName, "INT")) => row.selectDynamic[Int](colName)
        case (row, (colName, "BIT")) => row.selectDynamic[Boolean](colName)
        case (row, (colName, "VARCHAR")) => row.selectDynamic[String](colName)
    }.toClass(name = r.name)
    val storeCls = new ColumnStore[Scanner](recordCls)
    val store = field(code"${storeCls.make}()")
    val loadFrom = method(code" "(in: InputStream) => {
        val reader = new BufferedReader(in)
        while (reader.ready())
            $store.${storeCls.add}(new Scanner(reader.readLine()))
    }" ")
    val dumpInto = method(code"(out: OutputStream) => ...")
}

```

Triple-quotes in Scala are used to write multi-line quotations. For brevity, we omit the implementation of `dumpInto`, which simply iterates over the table's indices and fields, accesses them using `getFieldAt`, and prints out their `toString` representations.

Internally, our database engine will keep a mapping from each known table name to the associated staged class *instance* storing that table. The table instance is obtained by generating the code for instantiating the staged class using its factory's `make` method, and compiling that code on the fly:

```
class DatabaseInstance {
  val tables: Map[String, Table] = Map.empty
  def createTable(r: TableCreationRequest) = {
    val maker: Unit => Table =
      new ColStoreTable(r).make.compile
    tables(r.name) = maker()
  }
}
```

9.3 Compiling Queries On The Fly

Let us imagine we have a `Person` database table, and a user sends a query to find the age of the oldest person, as in `'SELECT MAX(p.Age) FROM Person p.'` We will first send this query to a parser and then to a query planner (not described in this chapter), which will return a query plan of the form `new Aggregate("Person", "age", "MAX")`. How can we implement such a query in an efficient way? One option is extending `Table` to include methods for iterating over the record indices, and getting the appropriate fields:

```
trait Table {
  ... // as before
  type RecordHandle
  def iterator: Iterator[RecordHandle]
  def getField(fieldName: String, r: RecordHandle): Any
}
```

But we can see two immediate problems with such approach: First, it is untyped (`getField` returns `Any`), which will result in dynamic tests and casting to the correct types, as well as boxing of primitive values, in addition to not being type safe. Second, the iteration is driven by an `Iterator` abstraction, relying on virtual calls, which we would like to avoid.

Thankfully, this is a case where the great power of *multi-stage* programing can be of help.

9.3.1 An Additional Stage for Compiling Queries

We start by defining an `Iter[A]` type that behaves like an iterator, but is defined in terms of a pair of (*has-next*, *get-next*) functions, so that these can be inlined aggressively by Squid:

```
type Iter[A] = (() => Boolean, () => A)
```

Then, we extend the `Table` interface to include: (1) a reference to the record class `elemCls` whose parameters are stored in the table; (2) an abstract `RecordHandle` type to represent logical “pointers” to the records stored in the table; (3) a way to create *the code* to iterate through the records using `Iter[RecordHandle]`; and (4) a type-safe `getField` method which extracts the value of a given field:

```
trait Table {
  ... // as before
  val elemCls: Class
  type RecordHandle
  def mkIter: Code[Iter[RecordHandle], {}]
  def getField[T](f: elemCls.Param[T]): Code[RecordHandle => T, {}]
}
```

We can now create an efficient implementation for queries such as MAX aggregations. The `mkMax` function below creates a program fragment for computing the current maximum of some field `f` in a table `tbl`:

```
def mkMax(tbl: Table)(f: tbl.elemCls.Param[Int]): Code[() => Int, {}] =
  code " " " () => {
    val (hasNext, getNext) = ${tbl.mkIter}
    var max = Int.MinValue
    while (hasNext()) {
      val h = getNext()
      val v = ${tbl.getField(f)}(h)
      if (v > max) max = v
    }
    max
  } " " "
```

In order to call this function, we will need to convert the `Aggregate` query object into the table and typed field arguments we want to pass to `mkMax`:

```
def mkAggr(db: DatabaseInstance, aggr: Aggregate): () => Int = {
  val tbl: tbl.elemCls.Param[_] = db.tables(aggr.tableName)
  val f = tbl.elemCls.parameters.find(_.name == aggr.fieldName).get

  assert(aggr.opName == "MAX", "Unsupported aggregation operator")
}
```

```
if (!(f.tpe <:< codeTypeof[Int])) // '<:<' checks subtyping at runtime
  throw new Exception("Cannot perform MAX on this field type")
val fInt = f.asInstanceOf[tbl.elemCls.Param[Int]]
// ^ there is no way to check this statically

mkMax(tbl)(fInt).compile
}
```

Naturally, this is a fallible process, as the “stringly-typed” Aggregate object could contain information that is inconsistent with our data schema. Proper validation and error reporting to the user would normally need to be implemented, in a mature real system.¹

Finally, to accommodate for the change in Table interface, the ColStoreTable class has to be adapted to include the following new definitions:

```
class ColStoreTable(r: TableCreationRequest) extends ... with Implements[Table] {

  @crossStage // this allows cross-stage references to recordCls
  val recordCls = ... // this is defined as before

  ... // other definitions as before

  val elemCls: Field[recordCls.type] =
    field(code"recordCls") // cross-stage reference to recordCls

  val RecordHandle = typeDef[Int] // we use an integer index as the handle

  val getSize: Method[() => Int] = method(
    code"() => $store.${storeCls.size}()" )

  val mkIter = method(code" "" {
    var idx = 0
    (() => idx < $getSize, // do we still have handles left?
    () => {
      val res = idx
      idx += 1
      res // next handle to use
    })
  }" "" )

  val getField
```

¹An alternative would be to use a type-safe embedded DSL as the frontend query language, as in Section 9.4 and Chapter 10.

```

: Method[[T] => recordCls.Param[T] => Code[Int => T, {}]]
= method({
  @crossStage
  val getFieldImpl = [T] => (f: recordCls.Param[T]) =>
    code"(idx: Int) => $store.${storeCls.getFieldAt(f)}(idx)"
  code"getFieldImpl"
})
}

```

The `typeDef` method (which we have omitted to present before) is used to provide a **type** synonym in the generated class. Here, we use it to implement the abstract **type** `RecordHandle` declared in the `Table` abstract class.

Three significantly non-trivial things happen here:

- First, in the definition of `elemCls`, we make a direct reference to the local `elemCls` field from within a `code` quasiquote — notice how it is not escaped; this is permitted because Squid supports cross-stage persistence [Taha and Sheard, 2000], which means that the compiled version of a `ColStoreTable` instance will contain a field that refers back to the `recordCls` field of the original staged `ColStoreTable` class!
- Second, `getField` is a first-class generic method (notice the leading `[T] =>` parameter); indeed, the abstract method required by `Table` is a polymorphic one.
- Third, `getField` *also* makes a cross-stage reference, to the `getFieldImpl` value; this is what will allow generating code for accessing the table’s fields even after we have compiled this instance of `ColStoreTable`.

9.4 An Embedded DSL for Data Definitions and Queries

In the previous sections of this chapter, we presented a staged database system designed to interact with users through plain SQL queries. While common and practical, such an interface is often suboptimal: its dynamic typing discipline makes it error-prone, and its expressiveness is limited to the constructs of SQL, an old and inflexible language which notably lacks abstraction capabilities.

Expressing queries through an EDSL has several advantages over plain SQL, including performance, type safety, and the reduction of the “impedance mismatch” between the application programming world (which uses general-purpose languages) and the database world (which uses query languages). Therefore, we would like to offer an EDSL frontend for our staged database system efforts.

In this section, we consider a better frontend to the staged database system. We describe how Squid can be used to reinterpret type-checked Scala constructs, turning them into internal

database-specific representations, to be compiled to efficient implementations by our staged engine.

Our approach transforms Scala class, field, and method definitions into a staged database representation at compilation time. The output of this process is the generation of a new database module containing specialized data structures and where each query method has been compiled to efficient database implementations.

9.4.1 Shallow DSL

The *shallow* DSL is the user-facing Scala library we expose to users, which they use to define their data types, data tables, and queries.

Data Definitions via Squid-Embedded Classes

To make our system easy and natural to use, we want to allow users to define the data types they want to store in the database as normal Scala classes, which may have parameters, fields, and methods. This way, we can reduce the impedance mismatch between applications written using functional and object-oriented code, and the databases used to store the corresponding data durably.

There are many ways of mapping Scala classes to database objects, and conversely. In object-relational mappings (ORMs), database accesses are often hidden behind method calls and field accessors, which is a leaky abstraction with many surprising performance characteristics. Here, we avoid such problems by requiring that the classes used to model the database be *private* to the database, so that communication with the outside has to be handled explicitly by users when defining queries. This approach is more aligned with the *data first* philosophy of functional programming.

As an example, we define below a database called `MyDatabase` which stores instances of two data types `Person` and `Job`:

```
@embed
object MyDatabase {
  private class Person(val name: String, var age: Int, var job: Option[Job])
    extends Record {
    def isMinor = age < 18
  }
  private class Job(val enterprise: String, val salary: Int) extends Record

  // table definitions follow...
}
```

To make the class, field, and method definitions inside `MyDatabase` available to Squid, we use

the `@embed` annotation (also used in Section 5.2 to retrieve the implementations of user-defined methods). The `@embed` annotation lifts Scala classes and objects into corresponding staged class representations, which can be directly manipulated, via Squid, by our staged database compiler.

Database Table Definitions

The basic interface for defining tables and manipulating table entries is the following:

```
class Table[T] {  
  def all: TableView[T]  
  def insert(el: T): Unit  
  def delete(el: T): Unit  
  def insertAtId(id: Long, el: T): Unit  
  def deleteAtId(id: Long): Unit  
  def getAtId(id: Long): Option[T]  
}
```

We use it to create one table for each data type in our database:

```
@embed  
object MyDatabase {  
  // ...  
  
  private val persons = new Table[Person]  
  private val jobs = new Table[Job]  
  
  // query definitions follow...  
}
```

Calling `persons.all` returns a `TableView[Person]`, against which we can now express queries.

Database Query Definitions

We express queries following the following `TableView` interface:

```
class TableView[T] {  
  def count: Int  
  def aggregate[Res](init: Res, acc: (T, Res) => Res): Res  
  def foreach(f: T => Unit): Unit  
  def filter(pred: T => Boolean): TableView[T]  
  def map[R](f: T => R): TableView[R]  
  def flatMap[R](f: T => TableView[R]): TableView[R]  
  def join[R](other: TableView[R])(pred: (T, R) => Boolean): TableView[(T, R)]  
}
```

```
}
```

Note that this basic query DSL is patently incomplete; the most obviously missing piece is a key-based aggregation operator (GROUP BY in SQL). In Chapter 10 we see a nice approach for expressing such queries in a very general way, and for optimizing them (Section 10.7).

For example, we define below a `loadFromFile` query for loading person and job data from CSV files, a `numberOfMinors` query for counting the number of minor people who have jobs, and a `deleteJobless` query for deleting people who do not have jobs:

```
@embed
object MyDatabase {
  // ...

  def loadFromFile(personsFile: String, jobsFile: String): Unit = {
    for (job <- Source.fromFile(jobsFile).getLines) {
      val Array(id, ent, sal) = line.split(',')
      jobs.insertAtId(id.toLong, new Job(ent, sal.toInt))
    }
    for (line <- Source.fromFile(personsFile).getLines) {
      val Array(name, jid) = line.split(',')
      val job = if (jid == "NULL") None
      else Some(
        jobs.getAtId(jid.toLong)
          .getOrElse(throw new Exception("Job not found: " + jid)))
      persons.insert(new Person(name, job))
    }
  }

  def numberOfMinors(): Int =
    persons.all.filter(_._2.isMinor).count

  def deleteJobless(): Unit =
    persons.all.filter(_._2 == None).foreach(persons.delete)
}
```

Generating the Staged Database Code

Finally, to get efficient runnable code out of the `MyDatabase` definition above, one has to explicitly ask for `Squid` to generate the corresponding code, which is done with the invocations below, which can for example be called as part of the project's build:

```
val MyDatabaseStaged = new StagedDatabase[MyDatabase]
MyDatabaseStaged.generateCode("generated/MyDatabase.scala")
```

Squid Intermediate Representation Setup

To set up the Squid intermediate representation required to manage the code of the user's queries, we define the IR object below. We make the intermediate representation extend the standard Squid SimpleANF representation in order to benefit from the automatic `val`-binding of nontrivial expressions, which will come in handy when lifting queries (Section 9.4.4):

```
object IR extends squid.ir.SimpleANF
  with squid.ir.StandardEffects
  with squid.ir.ClassEmbedder
  with squid.lang.ScalaCore
```

We also extend the `ClassEmbedder` trait in order to enable the Squid `@embed` helper macro, and `StandardEffects` in order to make the ANF representation leverage Squid's basic effect system for standard Scala constructs, such as primitives, primitive operations, and constructs from the standard library (see Section 4.4.2).

9.4.2 Internal Representation of the Database

The `StagedDatabase` class deals with representing staged databases and their queries, and with compiling them to lower level efficient code. We give the basic definitions of `StagedDatabase` below, and describe the other crucial parts of its design in the rest of this section.

```
class StagedDatabase[DB: CodeType] {

  /** Ctx represents the scope of this database; it will be used as the context
   * type for queries that make references to this database's tables. */
  type Ctx = cls.Ctx

  val db_cls: squid.Class[DB] = squid.classOf[DB]

  /** The representation of a table that lives in this staged database. */
  class TableRep[T0: CodeType](val cls: squid.Class[T0]) {
    type T = T0
    val T = codeTypeOf[T]
    val columns = cls.fields.map(f =>
      /* initialization of column information... */)
    val functions = cls.methods.map(f =>
      /* initialization of function information... */)
  }

  val tables = db_cls.classes.map(cls =>
    new TableRep(cls))

  /** The representation of a query expressed in this staged database. */
```

```
class Query[T: CodeType](val name: String, val cde: Code[T, Ctx]) {
  lazy val rep = liftQuery(cde)
}
val queries = db_cls.methods.map(m =>
  new Query(m.name, m.etaExpand))

// ...many more definitions elided...
}
```

The `TableRep` class is used to represent the tables of the database; it is created from the classes defined in the main database class `DB` — each class is treated like a table, whose columns correspond the class' fields, and with a set of functions to operate on the rows of the table, corresponding to the class' methods.

The queries defined in the database are gleaned from the top-level methods of the main database class `DB`, and stored as plain code inside the `Query` representation. The next step is to lift these code representations into proper query representations.

9.4.3 Query Representation

Queries are represented using the `QueryRep` generalized algebraic data types, for which we give a partial definition below. Notice that `QueryRep` has its own context parameter `C`, as queries may be nested within the local scopes of other queries:

```
/** Internal representation of database queries. */
sealed abstract class QueryRep[T: CodeType, C] {
  type Res = T
  implicit val Res = codeTypeOf[T]
}
```

We use local type synonyms (like `type Res = T` above), and an implicit declaration of the same name (like the `implicit val Res` above) in order to facilitate usages of the data type: given a `QueryRep` value `qr`, it allows one to **import** `qr.Res`, which brings into scope both a `Rep` type synonym and a `Rep` implicit type representation for it. Moreover, one can easily rename these symbols on import, as in `import qr.{Res => Res2}` to avoid name clashes.

Query DSL Constructors

Most of the constructors of this `QueryRep` GADT closely mirror the syntax of the shallow DSL and are unsurprising; we give them below for reference:

```
case class All[T: CodeType, C](tbl: TableRep[T])
  extends QueryRep[TableView[T], C]
```

```
case class Filter[T: CodeType, C]
  (q: QueryRep[TableView[T], C], pred: QueryRep[T => Boolean, C])
  extends QueryRep[TableView[T], C]

case class Map[T: CodeType, R: CodeType, C]
  (q: QueryRep[TableView[T], C], f: QueryRep[T => R, C])
  extends QueryRep[TableView[R], C] {
  type Row = T
  implicit val Row = codeTypeOf[Row]
}

case class Count[T: CodeType, C](q: QueryRep[TableView[T], C])
  extends QueryRep[Int, C] {
  type Row = T
  implicit val Row = codeTypeOf[Row]
}

case class Join[T: CodeType, R: CodeType, C]
  (q1: QueryRep[TableView[T], C], q2: QueryRep[TableView[R], C])
  extends QueryRep[TableView[(T, R)], C] {
  type Row1 = T
  implicit val Row1 = codeTypeOf[Row1]
  type Row2 = R
  implicit val Row2 = codeTypeOf[Row2]
}

case class Aggregate[T: CodeType, Res: CodeType, C]
  (q: QueryRep[TableView[T], C], init: QueryRep[Res, C],
   acc: QueryRep[(T, Res) => Res, C])
  extends QueryRep[Res, C] {
  type Row = T
  implicit val Row = codeTypeOf[Row]
}

case class Foreach[T: CodeType, C]
  (q: QueryRep[TableView[T], C], f: QueryRep[T => Unit, C])
  extends QueryRep[Unit, C] {
  type Row = T
  implicit val Row = codeTypeOf[Row]
}
```

Notice that we did not include a predicate in the definition of the `Join` constructor; this is because for regularity we represent join calls in the shallow DSL as a composition of `Join` and `Filter` in the internal `QueryRep` representation.

Scala Code Constructors

In addition to constructors mirroring the query DSL, we have several `QueryRep` cases to represent normal Scala operations whose semantics we handle explicitly, as they may interact with the semantics of the other query operators. For instance, we have `QueryReps` for **while** loops, lambda expressions, **val** bindings, and effectful statements.

```
case class While[T: CodeType, C]  
  (cond: QueryRep[Boolean, C], e: QueryRep[T, C])  
  extends QueryRep[Unit, C] {  
  type Val = T  
  implicit val Val = codeTypeOf[T]  
}
```

The `Lambda` constructor, which represents function values as `QueryReps`, has an interesting typing challenge: the type of its body is dependent on the type of the `Variable` bound by the lambda. Ideally, we would like to write the following definition:

```
case class Lambda[R: CodeType, T: CodeType, C]  
  (param: Variable[R])(body: QueryRep[T, C & param.Ctx])  
  extends QueryRep[R => T, C] {  
  type Arg = R  
  implicit val Arg = codeTypeOf[Arg]  
  type Ret = T  
  implicit val Ret = codeTypeOf[Ret]  
}
```

However, Scala 2 does not currently allow class parameters to depend on other parameters of the same class, so the type of body, `QueryRep[T, C & param.Ctx]`, is rejected by the compiler because of its dependency on the `param` class parameter.

There are several ways of solving the problem. One way is to use a type parameter `PCtx` to represent the context type of the variable, and then request that the variable's actual context match that specific context, using a `{ type Ctx = PCtx }` type refinement:

```
case class Lambda[R: CodeType, T: CodeType, C, PCtx]  
  (param: Variable[R] { type Ctx = PCtx }, body: QueryRep[T, C & PCtx])  
  extends QueryRep[R => T, C] {  
    // Arg and Ret defined as above  
}
```

An alternative is to use an abstract class for `Lambda` — leveraging the fact that class fields may have arbitrary dependencies between each other — and to provide a convenience constructor for the abstract class:

```
abstract class Lambda[R: CodeType, T: CodeType, C] extends QueryRep[R => T, C] {
```

```

    val param: Variable[R]
    val body: QueryRep[T, C & param.Ctx]
    // Arg and Ret defined as above
  }
  def Lambda[R: CodeType, T: CodeType, C]
    (_param: Variable[R])(_body: QueryRep[T, C & _param.Ctx]) =
    new Lambda[R, T, C] { val param: _param.type = _param; val body = _body }

```

Note that the `_param.type` ascription on the `param` field is crucial: without it, Scala widens the type of `param` to just `Variable[R]`, and the relationship of this value with `body` is lost.

Both of these approaches work fairly seamlessly in practice; it is only a matter of style which one to prefer. We adopt the former approach, as it is slightly more concise.

The Nested constructor below represents a `val` binding if the `x` variable occurs in `rest`, or an effectful statement otherwise:

```

case class Nested[R: CodeType, T: CodeType, C, XCtx]
  (x: Variable[R] { type Ctx = XCtx }, value: QueryRep[R, C],
   rest: QueryRep[T, C & XCtx])
  extends QueryRep[T, C] {
    type Val = R
    implicit val Val = codeTypeOf[Val]
  }

```

Finally, we have a `QueryRep` constructor for uninterpreted pieces of *plain Scala* code fragments, to be executed directly and without any query compilation applied to them:

```

case class PlainCode[T: CodeType, C](cde: Code[T, C])
  extends QueryRep[T, C]

```

9.4.4 Query Lifting

An essential task of our staged database design is that of *lifting* queries represented using normal Scala code — based on the shallow DSL — into proper query representations which are deeply embedded in Squid (presented in the previous subsection). This is done using the `liftQuery` method, shown below.

Remember from Section 9.4.1 that to facilitate the task of lifting queries, we use an ANF representation for representing the initial Squid code fragment, where each non-trivial expression is bound to a local `val` declaration. This is what allows the implementation of `liftQuery` to be so simple and straightforward — nested queries are automatically un-nested by ANF, so we only have to deal with queries bound by `val` bindings, not queries appearing in arbitrary subexpressions of general Scala expressions.

```
def liftQuery[T: CodeType, C](query: Code[T, C]): QueryRep[T, C] = query match {
  case code"val $x: $xt = $v; $rest: T" =>
    Nested(Some(x), liftQuery(v), liftQuery(rest))
  case code"$e; $rest: T" =>
    Nested(None, liftQuery(e), liftQuery(rest))
  case code"($param: $typ) => $exp: $ret" =>
    Lambda(param, liftQuery(exp))
  case code"while($cond) $e" =>
    While(liftQuery(cond), liftQuery(e))
  case code"($tbl: Table[$ty]).all" =>
    val tbl = findTable(tbl).getOrElse(liftingError("unknown table: " + tbl))
    All(tbl)
  case code"($view: TableView[$ty]).count" =>
    Count(liftQuery(view))
  case code"($view: TableView[$ty]).foreach($f)" =>
    Foreach(liftQuery(view), liftQuery(f))
  case code"($view: TableView[$ty]).filter($pred)" =>
    Filter(liftQuery(view), liftQuery(pred))
  case code"($view: TableView[$ty]).map[$tres]($f)" =>
    Map(liftQuery(view), liftQuery(f))
  case code"($view0: TableView[$ty0]).join($view1: TableView[$ty1])($pred)" =>
    Filter(Join(liftQuery(view0), liftQuery(view1)),
      liftQuery(code"(ab: ($ty0, $ty1)) => $pred(ab._1, ab._2)"))
  case code"($view: TableView[$ty]).aggregate[$tres]($init, $acc)" =>
    Aggregate(liftQuery(view), liftQuery(init), liftQuery(acc))
  case _ =>
    val cde = adaptCode(query)
    PlainCode(cde)
}
```

The `adaptCode` function, whose detailed implementation we omit in this presentation, is applied on Scala code fragments which do not represent queries; it has two purposes:

- To transform all field accesses and method calls performed in these Scala code fragments into the form they will need to have in the final generated code. In our approach, we represent the stored user data explicitly, without wrapping it into a class — and possibly in formats such as the column store representation — so the methods present in the original class defining the stored data types will be lifted as top-level functions, and their calls need to be adapted by this `adaptCode` function.
- To check that no queries are nested somewhere we cannot lift them, for example inside a lambda passed to an unhandled (unlifted) function. This way, if users mistakenly use the wrong constructs and write queries which cannot be interpreted and compiled

properly by our engine, a clean error is reported — as opposed to compiling broken or suboptimal code, which would lead to surprising performance.

Once we have queries represented in the form above, we can apply some rewritings on them, and then turn them into *query plans*, which represent lower-level query implementation strategies.

9.5 Basic Optimization and Planning for Queries

In this section we briefly review how queries are optimized by rewriting, and then planned before being turned into efficient low-level code.

9.5.1 Query Rewriting

The primary goal of query rewriting is to make the represented queries more regular and uniform, as well as to make them amenable to better planning (next subsection), allowing them to use efficient implementations.

Simple Query Optimizations

There are many rewritings performed by traditional query engines to improve the efficiency of query implementations.

For instance, we usually want to move the filter operations as close to the data source as possible, to avoid as many intermediate computations as possible. Given a query such as `Filter(Map(xs, f), pred)`, if the `f` computation is cheap (e.g., a field projection), we usually want to rewrite the query to `Map(Filter(xs, x => pred(f(x))), f)`.

Moreover, we want to merge filter operations together so they can later be properly scrutinized when later trying to extract join predicate (see Section 9.5.3).

There are more tricky cases, where we need to determine which parts of a filter predicate contain references to a free variables and which do not; for instance, consider the query `Filter(Join(xs, ys), pred)`. It is very often possible to separate the parts of `pred` which depends on `x`, on `y`, and on both, after which we can move the first two to the respective `xs` and `ys` subqueries. For instance, consider the predicate **case** `(x, y) => x > 0 && y > 0 && x > y`, which can be split into `x => x > 0`, `y => y > 0`, and **case** `(x, y) => x > y`. Like all operations involving bindings, this is usually an error-prone optimization to perform; thankfully, Squid's advanced scope-safe abstractions guarantee that we do not extrude variable references from predicates, removing a whole class of errors in the design of query optimizers.

Join Streamlining

In Scala, **for** comprehensions are implemented as compositions of `map`, `flatMap`, and `filter` method calls. This is a very expressive and flexible framework, but it can often be turned into a more restricted (and more optimizable!) representation. Consider the following query:

```
for (a <- as; b <- bs; if p(a, b)) yield f(a, b)
```

This is automatically rewritten by the Scala type checker into:

```
as.flatMap(a => bs.flatMap(b => (a,b)))  
  .filter(ab => p(ab._1, ab._2))  
  .map(ab => f(ab._1, ab._2))
```

In the case when `bs` does not refer to `a`, the same query can be expressed in a much more precise and direct way as:

```
as.join(bs)(p)  
  .map(ab => f(ab._1, ab._2))
```

The query planner will typically be able to implement the latter in a much more efficient way, including *asymptotic* efficiency gains, using different join implementation strategies based on the properties of the source collections and on the join predicate `p`.

We can perform that rewriting automatically based on the `QueryRep` representation. Again, this requires careful consideration about scopes and free variables, where Squid's statically-typed contexts are extremely useful in helping to prevent common mistakes.

9.5.2 Query Plans

The query plan representation, `QueryPlan`, is another staged data type. It is slightly lower-level than `QueryRep` and contains more information on how each subquery is meant to be implemented. Query plans have a `getCode` method which is used to emit the actual Scala code which implements each operator:

```
sealed abstract class QueryPlan[Res: CodeType, -C] {  
  def getCode: Code[Res, C]  
}
```

Iteration Plans

The simplest plan is of course the execution of plain Scala code (corresponding to the `PlainCode` constructor of `QueryRep`):

```
case class PlainExec[Res: CodeType, C](cde: Code[Res, C])
```

```

    extends QueryPlan[Res, C] {
    def getCode: Code[Res, C] = cde
    }

```

We separate query plans between two categories: *general* query plans, and *iteration* query plans. The latter represents a particular form of queries which iterates over an existing collection or subquery; it is given below:

```

/** The plan for some iteration as part of a bigger query plan. */
sealed abstract class IterationPlan[Row: CodeType, -C]
    extends QueryPlan[TableView[Row], C] {

    def push[C0 <: C](step: Code[Row => Boolean, C0]): Code[Unit, C0]

    def foreach[C0 <: C](step: Code[Row => Unit, C0]): Code[Unit, C0] =
        push(code{ row: Row => $(step)(row); true })

    def getCode: Code[TableView[Row], C] = code" " "
        val buff = mutable.Buffer.empty[Row]
        ${ foreach(code" (row: Row) => buff += row" ) }
        TableView.fromBuffer(buff)
    " " "
}

```

IterationPlan has a push method for generating code to push values produced by the plan into some callback function, until the callback returns false. The foreach method is just syntax sugar over push.

In the real system, we also have a slightly more complex (and often less efficient) pull method, which can be used for implementing certain more advanced query plans (like merge joins), but we do not show it here.

The simplest iteration plan is the one that simply scans the elements of an existing table:

```

case class Scan[Row: CodeType, C](src: TableRep[Row])
    extends IterationPlan[Row, C] {
    def push[C0 <: C](step: Code[Row => Boolean, C0]): Code[Unit, C0] = code" " "
        val (hasNext, getNext) = ${src.getIter}
        while (hasNext() && $step(getNext())) { }
    " " "
}

```

The `getIter` method from `TableRep` is used to iterate on the elements of the table; it will use the underlying staged class (following the approach described in Section 9.3.1).

Chapter 9. Application: A Staged Database Compiler

Next, we see iteration plans for selection (i.e., filtering) and projection (i.e., mapping):

```
case class Selection[Row: CodeType, C]
  (src: IterationPlan[Row, C], pred: QueryPlan[Row => Boolean, C])
  extends IterationPlan[Row, C] {
  def push[C0 <: C](step: Code[Row => Boolean, C0]): Code[Unit, C0] =
    src.push(code"(row: Row) => if (${pred.getCode}(row)) $step(row) else true")
}

case class Projection[Row: CodeType, RowRes: CodeType, C]
  (src: IterationPlan[Row, C], f: QueryPlan[Row => RowRes, C])
  extends IterationPlan[RowRes, C] {
  def push[C0 <: C](step: Code[RowRes => Boolean, C0]): Code[Unit, C0] =
    src.push(code"(row: Row) => $step(${f.getCode}(row))")
}
```

General Plans

The query plan for aggregating values is not an iteration plan, but it *uses* one as the source of the data that is being aggregated:

```
case class Aggregation[Row: CodeType, Res: CodeType, C]
  (src: IterationPlan[Row, C],
   init: QueryPlan[Res, C], acc: QueryPlan[(Row, Res) => Res, C])
  extends QueryPlan[Res, C] {
  def getCode: Code[Res, C] = code"""
    var res = ${init.getCode}
    ${ src.foreach(code"(row: Row) => res = ${acc.getCode}(row, res)") }
    res
  """
}
```

Below is a plan for computing the result of another query first, and then executing the rest of the query (which corresponds to the Nested query representation):

```
case class NestedPlan[R: CodeType, T: CodeType, C, XCtx]
  (x: Variable[R] { type Ctx = XCtx }, value: QueryPlan[R, C],
   rest: QueryPlan[T, C & XCtx])
  extends QueryPlan[T, C] {
  def getCode: Code[T, C] = {
    val restCode = rest.getCode
    x.tryClose(restCode) match {
      case Some(closed) => code"${value.getCode}; $closed"
      case None         => code"val $x = ${value.getCode}; $restCode"
    }
  }
}
```

Figure 9.1 – query plan for a nested-loop join

```

case class HashJoin[Row1: CodeType, Row2: CodeType, K: CodeType, C]
  (it1: IterationPlan[Row1, C], it2: IterationPlan[Row2, C],
   getK1: Code[Row1 => K, C], getK2: Code[Row2 => K, C])
  extends IterationPlan[(Row1, Row2), C] {
def push[C0 <: C](step: Code[(Row1, Row2)) => Boolean, C0])
    : Code[Unit, C0] = code " "
  val hm = mutable.HashMap.empty[K, mutable.Buffer[Row1]]
  ${ it1.foreach(code " "(row1: Row1) => {
    val k = $getK1(row1)
    val buff = hm.getOrElseUpdate(k, mutable.Buffer.empty)
    buff += row1
  } " " ) }
  ${ it2.push(code " "(row1: Row1) => {
    val k = $getK2(row2)
    val buff = hm.get(k)
    if (buff.isDefined) buff.get.forall { row1 =>
      $step((row1, row2))
    } else true
  } " " ) }
  " " "
}

```

Figure 9.2 – query plan for a hash join on some key K

```

/** Turn a basic query into a query plan. */
def planQuery[T, C](rep: QueryRep[T, C]): QueryPlan[T, C] = rep match {
  case c: Count[_ , C] =>
    import c.Row // to get the implicit type representations in scope
    Aggregation[Row, Int, C](planIteration(c.q), PlainExec(code"0"),
      PlainExec(code"(row: Row, acc: Int) => acc + 1"))
  case a: Aggregate[_ , _ , C] =>
    import a.{Row, Res}
    Aggregation[Row, Res, C](planIteration(a.q), planQuery(a.init),
      planQuery(a.acc))
  case f: Foreach[_ , C] =>
    import f.Row
    QueryForEach[Row, C](planIteration(f.q), planQuery(f.f))
  case w: While[t, C] =>
    import w.Val
    QueryWhile(planQuery(w.cond), planQuery(w.e))
  case f: Lambda[targ, tres, C, xCtx] =>
    import f.{Arg, Res, Ret}
    Closure(f.param, planQuery(f.body))
  case l: Nested[tx, tres, C, xCtx] =>
    import l.{Val, Res}
    NestedPlan(l.x, planQuery(l.value), planQuery(l.rest))
  case c: PlainCode[t, C] =>
    import c.Res
    PlainExec(c.cde)
  case All(_) | Map(_, _) | Filter(_, _) | Join(_, _) =>
    planIteration(rep)
}

```

Figure 9.3 – Basic general query planning (planIteration is shown in Figure 9.4.)

```
/** Turn an iteration query into a query plan. */
def planIteration[T: CodeType, C](rep: QueryRep[TableView[T], C])
  : IterationPlan[T, C] = rep match {
case All(tbl) => Scan(tbl)
case map: Map[typ, tres, C] =>
  import map.Row
  Projection(planIteration(map.q), planQuery(map.f))
case join: Join[t1, t2, C] =>
  import join.{Row1, Row2}
  NestedLoopJoin(planIteration(join.q1), planIteration(join.q2))
case Filter(view, pred) =>
  def defaultImpl = Selection(planIteration(view), planQuery(pred))
  view match { // try to plan more elaborate join implementations:
    case join: Join[t1, t2, C] =>
      import join.{Row1, Row2}
      extractJoinPredicate(pred) match {
        case hj: HashJoinPred[k] =>
          import hj.K
          val impl = HashJoin(planIteration(join.q1),
            planIteration(join.q2),
            hj.getK1, hj.getK2)
          hj.predRest match {
            case Some(pred) =>
              Selection(impl, planQuery(pred))
            case None => impl
          }
        case _ => defaultImpl
      }
    case _ => defaultImpl
  }
}
```

Figure 9.4 – Basic iteration query planning.

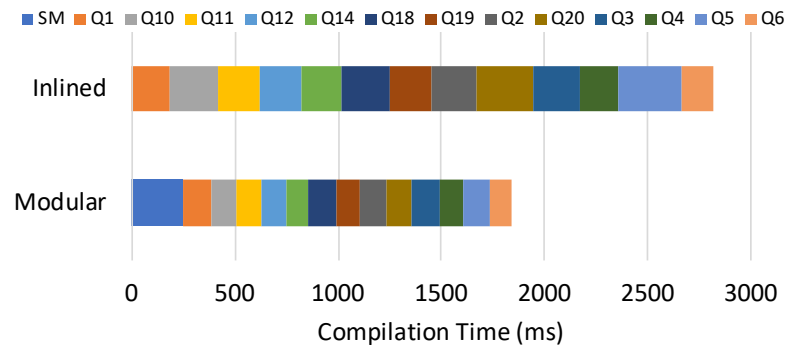


Figure 9.5 – Compilation times for TPCCH queries, comparing a “modular” approach where the common storage manager (SM) infrastructure used by the queries is factored into a separately-compiled file, and an “inlined” approach where the required infrastructure is duplicated in each query.

Figure 9.3 shows how general query plans are created from query representations; there is nothing really surprising about it. More interestingly, we show how to plan iteration queries in Figure 9.4. In order to select efficient join implementations when available, at the time of scheduling a `Filter` query, we check if the underlying view is a `Join` query; if so, we try to extract a useful join predicate from it using the `extractJoinPredicate` (whose implementation, based on simple Squid pattern matching, we do not show here). This function may successfully extract a `HashJoinPred` predicate, which contains the information we need to construct a hash join plan from it. More join implementation strategies can be devised and planned using this strategy — one simply has to add the corresponding join plans, adapt the `extractJoinPredicate` function appropriately, and handle the new cases in `planIteration`.

9.6 Evaluation

It would be hard to evaluate the performance of our limited database system against existing approaches. Indeed, it would be unfair to compare it with mature industrial engines for the JVM, as these are designed to handle a lot more scenarios and use cases, and as a result they will likely have significant overhead compared to our simplistic prototype.

Instead, here we compare the original version of an existing *static query compiler* previously built with Squid (call it SQC), to a new modularized version built with Squid + staged classes (call it MSQC). Previously, SQC generated all the code necessary for dealing with storage management of the database tables *with each query*, which resulted in significant recompilation costs. Moreover, the data structures were generated using ad-hoc mechanisms based on cobbling strings together, which had proven brittle and limiting. We have reimplemented and modularized the data structure manipulation of SQC with staged classes (giving MSQC), so that the storage management code is generated in a reliable way and is compiled only once, and so that each query can refer to it safely.

Both SQC and MSQC are configured to generate C code at the end of the compilation pipeline,

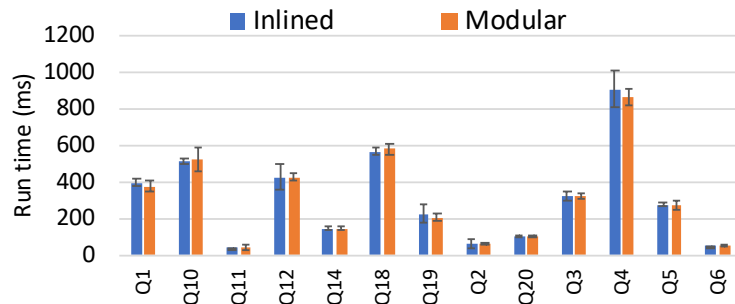


Figure 9.6 – Running times for the TPC queries in Figure 9.5, comparing the “modular” and “inlined” approaches.

and to offload C compilation to the Clang compiler (version clang-900.0.39.2). Figure 9.5 compares the cumulated compilation times of the generated C code of a set of TPC queries [Transaction Processing Performance Council, 1999] for SQC (“Inlined”) and MSQC (“Modular”). We have also verified that the modularization of the compiled query code did not, in fact, introduce any performance degradation. Figure 9.6 shows that the performance of each query is comparable for SQC (“Inlined”) and MSQC (“Modular”).

All the measurements in this chapter were made on an Intel Core i7 CPU running at 3.5GHz, 16GB of DDR3 RAM at 2133Mhz, with macOS 10.13.6.

9.6.1 Related Work

There are many database libraries for high-level languages like Scala, such as Slick, Quill, Doo-bie, etc., which all have creative ways of representing and handle database queries. However, the vast majority (if not all) of these efforts are focused on accessing external databases. This project goes much beyond that goal, as it tries to actually implement a database system from first principles, in a high-level language, using metaprogramming to avoid for paying the cost of abstraction.

As a corollary, this tight language integration between our query language and our database implementation makes our system much more flexible (not restricted to the only data types, operations, and queries supported by SQL). The goal is also to allow users to customize the database runtime system (query optimization capabilities, query operator implementations, storage management, etc.), not merely to provide a query-DSL-to-SQL translation.

Finally, contrary to LINQ and other similar approaches, the “expression trees” we lift from query expressions are typed, thanks to Squid’s support for type-safe metaprogramming. This means the implementation of the query lifter is inherently safer, as type and scope errors cannot be introduced by incorrect manipulations of program fragments.

9.7 Conclusion

In this chapter, we have sketched a real-world-inspired staged database compiler. We made use of staged classes in order to generate efficient reusable data structures at run time. The metaprogram we have shown uses three successive stages: in the first stage, we compile the containers used to hold the database data in an efficient way; in the second stage, we compile individual queries making use of these containers (the queries are assumed to be received later on, over the network); the last stage corresponds to actually running the queries. We used cross-stage persistence in order to propagate the type information needed to compile queries after having compiled the data containers we want to query. Finally, we presented a more advanced alternative frontend for our system, making full use of Scala's DSL design capabilities and Squid's type-safe analytic metaprogramming.

10 Comprehending Monoids with Class

Powerful high-level languages like Scala and Haskell offer the ability to define expressive embedded domain-specific languages (EDSL) for writing database queries in them, leveraging their expressive type systems.

In the previous chapter, we presented a staged database system designed to interact with users through or an embedded Scala DSL inspired by relational query languages (in addition to its SQL frontend); however, there is a wide design-space for such query languages, and we have so far only explored a small corner of it.

This chapter presents the result of my search for the best DSL, to embed in languages like Scala or Haskell, for expressing queries over collections of data.

We revisit an old concept, *monoid comprehension*, and explore how to integrate it with a *functional programming* language with support for *type classes*, as well as how it compares to the more traditional monad comprehension approach. We argue that for expressing queries over collections of data, our embedding of monoid comprehension can be *more flexible*, *simpler*, *more efficient*, and *safer* than its monadic counterpart.

10.1 Background on Comprehension

This sections briefly recalls some background on the common ‘set comprehension’ syntax.

10.1.1 Origins

The comprehension syntax dates all the way back to Cantor’s work on set comprehension (circa 1874). A set comprehension is an expression of the form $\{ x \mid f(x) \}$, which denotes the set of all x for which $f(x)$ holds.

Unfortunately, this approach turned out to lead to inconsistencies, as it was too flexible. For

example, consider the set S defined as:

$$S = \{ x \mid x \notin S \}$$

What should be the truth value of the statement $S \in S$? If it is true, then by definition it must be false, and *vice versa*. (This is known as Russell’s paradox.)

With their ZFC theory, Zermelo (1908) and Fraenkel (1922) restricted the syntactic form of set comprehensions, so that the defined set elements be required to “range” over an existing set X , and that set X as well as the predicate f were not allowed to refer to the set being defined:

$$S = \{ x \mid x \in X, f(x) \} \quad \text{where} \quad S \notin \text{FV}(X) \cup \text{FV}(f)$$

This avoided the paradox, but required new axioms to be added to the theory, as a result of the reduction in the expressiveness of set comprehension.

On the other hand, restricting the scope of the comprehension syntax incidentally made it useful for programming. As early as the 1970s, programming languages started incorporating an analogous syntax for building *lists* of elements from preexisting lists. For example:

$$L = [(x, y) \mid x \leftarrow X, f(x), y \leftarrow Y, g(x, y)]$$

which iterates over two existing lists X and Y and produces a list of pairs, while filtering the output using predicates f and g .

This of course raises the question: “why just lists?,” and indeed Wadler [1990] showed that the comprehension syntax could be generalized to any monad. And since in Haskell, the input/output effect type IO is a monad, one could now conveniently write effectful expressions such as:

$$P = [() \mid x \leftarrow \text{getLine}, () \leftarrow \text{print } x]$$

where P is given type $\text{IO } ()$, the type of programs that performs side effects and then return the unit value $()$.

As an aside, a different syntax was soon preferred for such monadic expressions, the “do”-notation, which looked closer to the imperative programming constructs that it could be used to express:

$$\text{do } \{ x \leftarrow \text{getLine}; () \leftarrow \text{print } x; \text{return } () \}$$

10.2 Comprehension for Queries

We have known for a while that list comprehensions are a very natural way to express certain queries over relational databases [Trinder, 1992, Grust, 2004]. For example, consider the following SQL query:

`q0 = SELECT name, age FROM Persons WHERE age > 18`

...which can be expressed as the list comprehension:

$$q_0 = [(\text{name } p, \text{age } p) \mid p \leftarrow \text{Persons}, \text{age } p > 18]$$

However, not all SQL query features fit well in this encoding. For example, it is not easy to express grouping aggregations (the SQL `GROUP BY` form) and ordering (`ORDER BY` in SQL).

Thus, Jones and Wadler [2007] extended the syntax of list comprehension with generalizations of these two constructs, and Giorgidze et al. [2011] generalized them to all monadic comprehensions. Below is an example of that extended syntax (we explain it later in this chapter):

$$[f \ b \mid (a, b) \leftarrow xs, \text{ then group by } a \text{ using } g]$$

In parallel to all this, another interpretation of comprehension in the context of programming was devised by Fegaras and Maier [1995, 2000] who generalized list comprehension to *monoid* comprehension, instead of going for monads. Here is what that approach looks like:

$$++ \{ (x, y) \mid x \leftarrow X, f(x), y \leftarrow Y, g(x, y) \}$$

where `++` here is list concatenation. It indicates that we are building a *list* as a result of the comprehension, telling us how to combine intermediate results. We will explain in more detail how this works, and why I think it is often better at expressing queries than monad comprehension.

The goal of this chapter is to shed some light on the possibilities of monoid comprehension in the context of functional programming with type classes. I believe the approach has not received the interest it deserves from our community, partly because it had never been properly embedded within the type system of a functional language.

10.3 Why Monoid Comprehension?

Before we go further, let us quickly recall the semantics of monad comprehension and then explore that of monoid comprehension, based on some examples.

10.3.1 Semantics of List and Monad comprehension

Given a list comprehension such as:

$$[g \ x \ y \mid x \leftarrow xs, y \leftarrow ys, f \ x \ y]$$

the Haskell compiler will (at least conceptually¹) emit the following code:

```
q1 = concatMap (\x ->
    concatMap (\y ->
        if f x y then [g x y] else []
    ) ys
) xs
```

where `concatMap :: (a -> [b]) -> [a] -> [b]` takes a list, maps a list-returning function over it, and returns the concatenation of all the resulting lists.

The monadic generalization is very similar, but obviously uses monadic operators, which are strictly more general, instead of list operators:

```
q1 = (>>= \x ->
    (>>= \y ->
        if f x y then return $ g x y else mzero
    ) ys
) xs
```

The first thing to notice is that this form of comprehension requires *full homogeneity* between the sources being iterated, as well as the result being built. In the example above, *xs*, *ys*, and *q* must all be of the same Monad type *m*.

10.3.2 Embedding Monoid comprehension in Haskell

In this chapter, I propose an embedding of monoid comprehension in Haskell where **type class resolution** is used to *implicitly* resolve which monoid instance we want to use.

That is, instead of writing `++ { e | ... }` (for the list monoid) or `&& { e | ... }` (for the boolean 'and' monoid) as shown in the previous section, we will write just `{ e | ... }` and let the type checker *infer* which monoid interpretation to use based on the type of expression *e*.

This may seem like a very minor change, but I argue that it actually **unleashes the expressive power of monoid comprehension**. Indeed, not all monoids can be characterized by a single operator, as some monoid instances are *derived automatically* from more primitive ones. We will see interesting examples of that towards the end of this chapter.

10.3.3 Semantics of Monoid comprehension

Given the following hypothetical syntax for monoid comprehension:

$$q_2 = \{ g \ x \ y \mid x \leftarrow xs, y \leftarrow ys, f \ x \ y \}$$

¹Optimizing compilers like the Glasgow Haskell Compiler may, in practice, emit a fast recursive function to implement this list comprehension.

The translation would be very close to that of list (or monad) comprehension, except that instead of `concatMap` (or `>>=`) we use `foldMap`, and instead of wrapping the yielded expression into a singleton list (or `return`), we do not wrap it at all:

```
q2 = foldMap (\x ->
  foldMap (\y ->
    if f x y then g x y else mempty
  ) ys
) xs
```

where `foldMap :: (Foldable f, Monoid m) => (a -> m) -> f a -> m` takes a monoid-returning function, maps it over anything that can be folded, and returns the monoidal merge of all the results.

(Recall that a monoid is a type with an associative binary operation and an ‘identity’ element.)

The important thing to notice now is that contrary to the monadic interpretation, **this requires no homogeneity at all** between the different parts of the query: `xs` and `ys` can be different `Foldable` types, and `q` can be any `Monoid`!

This becomes more apparent if we examine the type of the query after parameterizing it with all its free variables:

$$\begin{aligned} \text{query } f \ g \ xs \ ys &= \{ g \ x \ y \mid x \leftarrow xs, y \leftarrow ys, f \ x \ y \} \\ \text{query} &:: (\text{Monoid } m, \text{Foldable } f_1, \text{Foldable } f_2) \Rightarrow \\ &\quad (a \rightarrow b \rightarrow \text{bool}) \rightarrow (a \rightarrow b \rightarrow m) \rightarrow f_1 \ a \rightarrow f_2 \ b \rightarrow m \end{aligned}$$

where we can see that `query` can be invoked with any combinations of two `Foldable` inputs and a `Monoid` output.

As a concrete example of monoid comprehension defined on heterogeneous types, the following query computes a *sum* from iterating over a *list* and a *set*:

$$\{ \text{Sum } (\text{length } y + x) \mid x \leftarrow [1,2,3], y \leftarrow \text{Set.fromList } ["a", "bb", "ccc"] \}$$

(In Haskell, the `Sum` “newtype” is used to wrap a numeric type giving it the monoid instance corresponding with summation — indeed, there are other possible monoid interpretations for these types, such as `Product`.)

10.3.4 Encoding

It is clear that the result of a monad comprehension `[e | c...]` that works on foldable inputs `c...` and a monoid result `e` (which is the case of most comprehensions on collections of data)

should correspond to the result of an equivalent monoid comprehension $\{ e \mid c... \}$, so we say that monoid comprehension “subsumes” monoidal monad comprehension.

On the other hand, we can encode a monoid comprehension with a list comprehension in a straightforward way. All we need to do is wrap the comprehension in a fold, and wrap each source in a `toList` (both from the `Data.Foldable` module):

$$\begin{aligned} q_3 &= \{ g \ x \ y \mid x \leftarrow xs, y \leftarrow ys, f \ x \ y \} \\ &= \text{fold } [g \ x \ y \mid x \leftarrow \text{toList } xs, y \leftarrow \text{toList } ys, f \ x \ y] \end{aligned}$$

so monoid comprehension offers no gain in terms of pure expressive power. But besides the slight reduction in syntactic overhead² and the added flexibility given by heterogeneity, it still has several advantages over monad comprehension, as we shall see below.

10.3.5 Space Efficiency

First of all, notice that using list comprehension to model database queries needlessly creates a lot of intermediate list data structures.

In the case of q_3 as defined in the previous section, not only will we create one extraneous list for each source collection (if it is not already a list) but we will also create the cartesian product of xs and ys as a list of size $\text{length } xs \times \text{length } ys$ ³ and fold the result into the monoid returned by g (though if that monoid is sufficiently lazy, the cartesian product list will not be fully materialized in memory, and will instead be progressively produced and consumed in constant space).

Compare that with the monoid comprehension approach, *which does not create any intermediate data structures at all* and is thus *asymptotically better* in terms of space complexity.

So, why not comprehend monoids directly?

10.4 SQL-style Grouping and Ordering

One of the nice things about monoid comprehension is that it gives us constructs for grouping and ordering for free, without any additions to the syntax. But first, let us explain how it is currently done in monad comprehension.

²It is also possible to encode monoid comprehension as a monad comprehension using the continuation monad and conversions from the input collections that wrap up a call to `foldMap` in the continuation.

³This is not actually always true: GHC is smart enough to produce efficient loops instead of calls to `concatMap` for list comprehensions, to avoid the creation of intermediate lists (`concatMap` is the official semantics of list comprehension, but the Haskell language specification does not mandate a particular desugaring). However, monad comprehensions cannot use the same trick, and have to pay the cost of `>>=`-based desugaring.

10.4.1 Grouping in monad comprehension

In query languages such as SQL, it is possible to *group* the result of a query by some fields from the original input. For example, the SQL query:

```
q4 = SELECT Avg(p.Salary) FROM Person p GROUP BY p.Age
```

computes, for each known age, the average salary of the persons of that age found in the table named ‘Person.’

The grouping construct of SQL does not fit well within the monadic interpretation of comprehension, which is why the list [Jones and Wadler, 2007] and monad [Giorgidze et al., 2011] comprehension syntaxes were extended to accommodate precisely that construct, along with a construct for ordering. For example, the SQL query above can now be written in Haskell as:

```
[ average (map salary p) | p ← persons, then group by age p using groupWith ]
```

where `groupWith` is the function used to do the grouping, and `average` simply computes the average of a list. The trick is that after the `then group by` expression (and also on the left-hand side of the `|`), the meaning of binding `p` changes from “the person currently iterated” to “the group of persons currently iterated,” where the groups are determined by the arguments given to `group by` and **using**. So while `p` has type `Person` right after the “`p ← persons`” generator, it *becomes a list* of type `[Person]` after the `group by` statement, and in the “`map salary p`” expression.

While clever, this is a nontrivial and quite idiosyncratic mechanism (I do not know of any other language in which a construct *modifies the meaning* of certain bindings without even mentioning them). And to add to the scoping conundrum, the variables defined in the comprehension are not in scope of the function passed to ‘`using`’, whereas they syntactically seem to be.

Moreover, if we want to remember the age associated with each average, we have to write:

```
q4 = [ (the a, average s) | p ← persons,
      let a = age p,
      let s = salary p,
      then group by a using groupWith ]
```

or slightly more succinctly:

```
q4 = [ (the age, average salary) | Person{age, salary} ← persons,
      then group by age using groupWith ]
```

where the `:: Eq a => [a] -> a` returns the head of a list and makes sure all the elements in that list are equal. This is somewhat unsatisfactory, because this is a partial function that throws runtime exceptions, and it is easy to get that part wrong, for example by writing `age $ the p` on the left-hand side of the `|` (where `p` is the `[Person]` variable), which crashes at runtime, instead of extracting `age` from the current `Person` (also named `p`), *on the right-hand side* of the `|` and then writing the `age` on the left-hand side, as above.

10.4.2 Grouping in monoid comprehension

Now, what does it take to have grouping in monoid comprehension? Perhaps surprisingly, the answer is ***nothing at all!*** In fact, ***grouping is trivial in monoid comprehension.***

Remember that all grouping does is aggregate elements into “buckets” based on some “key” extracted from the iterated elements. Well, this is what we would normally use a `Map` for — we just need a `Map` with an instance of `Monoid` that combines the values of shared keys based on their own `Monoid` (or `Semigroup`) instance, such as `Data.HashMap.Monoidal` provided by the `monoidal-containers` library.⁴

Here is how to write the example above using a monoid comprehension:

$$q_4 = \{ \text{Average } (\text{salary } p) \mid p \leftarrow \text{persons} \}$$

where `Average` is a newtype⁵ to aggregate values using their `Fractional` instance, and `groupBy = flip Data.HashMap.Monoidal.singleton` is just syntactic sugar for creating a singleton `Map` object, given a key and value.

Assuming the `salary` field of `Person` is a `Float`, the expression above results in a `HashMap Int (Average Float)` mapping each age to the average salary of the persons of that age.

10.4.3 Performance of grouping

According to the GHC specification,⁶ the monad comprehension encoding of q_4 essentially desugars to:

```
groupWith (\(age,salary) -> age) [ (age,salary) |
  Person{age,salary} <- persons ] >>= \ys ->
  case (fmap (\(age,salary) -> age) ys, fmap (\(age,salary) ->
    salary) ys) of
    (age,salary) -> return (the age, average salary)
```

⁴`monoidal-containers`: <https://hackage.haskell.org/package/monoidal-containers>

⁵the `Average` newtype is provided at <http://hackage.haskell.org/package/average-0.6.1/docs/Data-Monoid-Average.html>

⁶GHC specification of generalized list comprehension: https://downloads.haskell.org/~ghc/7.8.3/docs/html/users_guide/syntax-extns.html#generalised-list-comprehensions

Notice how this expression computes many intermediate lists. Worse, it *traverses the entire data a grand total of (at least) six times!* The traversals are done by the call to `groupWith`, the conversion of the input list to a list of tuples, the two `fmap` applications, and finally calls to `sum` and `average`.

On the other hand, the monoid comprehension form desugars to just:

```
foldMap (\Person{age,salary} ->
  Average salary `groupBy` age) persons
```

which computes its result in a *single list traversal* and creates no intermediate lists at all. Of course, we are building a `Map`, with overall complexity $n \cdot \log(n)$, but this complexity is also present in the monadic form since `groupWith` needs to pre-sort its input list.

Note that list fusion will likely *not* remove all the extraneous intermediate lists of the monadic form (even when specialized to lists), since it makes non-linear use of lists like `ys` and the argument to `groupWith`.

10.4.4 Generality of grouping

You may be thinking that the extended monad comprehension syntax is more flexible, because it allows us to define separate aggregations on the result, as in:

$$q_5 = [(\text{sum } x, \text{average } y) \mid (x, y, z) \leftarrow \text{ls}, \text{ then group by } z \text{ using groupWith}]$$

but in fact, this is trivially expressed in monoid comprehension as well, thanks to the fact that a tuple of monoids is also a monoid, as in:

$$q_5 = \{ (\text{Sum } x, \text{Average } y) \text{ `groupBy` } z \mid (x, y, z) \leftarrow \text{ls} \}$$

which, contrary to the monadic version, only *performs a single traversal* of the source data, accumulating the result of `Sum` and `Average` “in parallel.” In fact, there are other parallel monoid aggregations which *cannot actually be expressed with a pure monad comprehension* (using only the ‘group by’ extended syntax), such as:

$$\{ (\text{Sum } x \text{ `groupBy` } y, \text{Average } y \text{ `groupBy` } x) \mid (x, y) \leftarrow \text{ls} \}$$

Another interesting flexibility of the monad comprehension form is that one can use an arbitrary grouping function in the `using` clause, not just `groupWith`. This flexibility is also present in the monoid comprehension form, where we would simply use a `Map` type with a different `Monoid` instance as the result of a new `groupBy'` function.

10.4.5 Ordering

The extended monad comprehension syntax also has built-in support for processing the current results of the comprehension, which can be used for ordering it, or dropping some elements from it, etc. For example, the following expression first “drops” 1 element of its input list, groups the remaining elements computing their *sum*, and then “takes” the first two such grouped sums:

```
q6 = [ (the a, average s) | (x, y) ← xs,
      then drop 1,
      then group by y using groupWith,
      then take 2 ]
```

The monoid comprehension syntax does not have a built-in way of doing that, but it is easily encoded by directly applying the processing functions in question:

```
take 2 { Sum x `groupBy` y | (x, y) ← drop 1 xs }
```

On the other hand, we can again define a new SQL-like construct by simply wrapping things into a type with the right monoid instance. In the case of ordering, all we have to do is to use a monoidal map that orders its elements, such as `Data.Map.Monoidal` provided by the `monoidal-containers` package. We can define syntax sugar `'orderBy = flip Data.Map.Monoidal.singleton'`, and then use it as in:

```
q7 = { (count, [x] `orderBy` Down y) `groupBy` z | (x, y, z) ← xs }
```

which has type `HashMap z (Sum Int, Map (Down y) [x])`. This query iterates over the (x, y, z) element of xs , creating one group for each distinct z , and in each of these groups:

1. counts the number of elements in it (where `count = Sum 1`);
2. creates one list of x for each distinct y , ordering these lists by y in descending order (`Down` is a newtype that reverses the canonical order `Ord` of a type).

This query is interesting because it is *neither expressible in SQL nor in pure (extended) monad comprehension*, which demonstrates the versatility of using monoid instances to compose queries with various meanings.

10.5 Conclusions on Monoid Comprehension in Haskell

In summary, we have seen that monoid comprehension is a useful alternative to list or monad comprehension, especially in the context of expressing queries over collections of data. Indeed, in this specific context it is:

- **more flexible**, since it allows iterating over heterogeneous data sources without coercions while being able to produce any monoid result in one go, and since it can express advanced queries (that monad comprehension cannot directly) simply by composing monoid instances together;
- **simpler**, as it has a straightforward desugaring, and does not need any extensions for expressing SQL-like queries (we have grouping and ordering “for free”);
- **more efficient**, because it requires fewer traversals of the processed data and fewer intermediate collections — in fact exhibiting asymptotically better space efficiency;
- **safer**, as it does not require the use of partial functions like ‘the’ to perform basic tasks such as grouping while retaining the grouping key.

10.6 Generalized Monoid Comprehension in Scala

We have also implemented monoid comprehension in Scala, following precisely the same type-class-based approach as seen in the previous section in the context of Haskell.

In the example below, we demonstrate an embedding in Scala, whose **for**-comprehension syntax is not necessarily monadic and can easily be repurposed to accept a *monoidal* interpretation:

```
for { fname <- fileNames
      word <- streamFile(fname).characters.splitOn(' ')
      if word.nonEmpty }
yield ( avg(word.length.toDouble),
        count().groupedBy(word.toLowerCase) )
```

This query iterates over all the words contained in a set of files and aggregates the global average word length as well as per-word case-insensitive occurrence counts. This is desugared to a composition of `map`, `flatMap` and `filter`, which we have overloaded to aggregate monoids. For example, one signature of `map` is $(A \Rightarrow R) \Rightarrow As \Rightarrow R$ where R has to be a monoid and As has to be a finite source of A elements. The type inferred is `(Option[Avg[Double]], Map[String, NonZero[Nat]])`.

Canonical Semigroup	Associated Canonical Monoid	Properties
(NonZero[Nat], _ + _)	(Nat, _ + _, 0)	C
(List[T], _ ++ _)	(List[T], _ ++ _, Nil)	O F
(NonEmpty[Set[T]], _ union _)	(Set[T], _ union _, Set.empty)	C I F
(Max[Nat], _ max _)	(Max[Nat], _ max _, 0)	C I
(Max[Int], _ max _)	(Option[Max[Int]], _.flatMap(m=>m max _), None)	C I
(Streamed[T], _ concat _)	(Streamed[T], _ concat _, Streamed.empty)	L O
(Incr[Set[T]], _ concat _)	(Incr[Set[T]], _ concat _, Incr.empty)	I L O
(Map[K, NonZero[Nat]], _ merge _)	(Map[K, NonZero[Nat]], _ merge _, Map.empty)	C F

Table 10.1 – Some example canonical semigroup instances, their associated canonical monoid forms, and their properties. Where C = commutative, I = idempotent, L = lazy, and for data sources O = ordered, F = finite.

10.6.1 The Full Monoid Comprehension Calculus

There is more to the monoid comprehension calculus (MCC) than the monoid comprehension syntax. Crucially, not all combinations of monoids are normally allowed in a given monoid comprehension. For example, if one of the generators is a set, the result type cannot be a list, because that would make the semantics of the query dependent on the order in which the set is iterated (which is unspecified).

Restrictions imposed on the allowed combinations of monoids not only make query semantics deterministic, but also give more freedom to the query engine, which has more options for parallelizing query executions.

In the rest of this section, we briefly describe a generalization of MCC (including the restrictions on monoid combinations), as part of our Scala embedding — to the best of our knowledge the first such typed embedding.

10.6.2 Semigroups and Canonical Monoids

Reasoning exclusively about monoids is too restrictive; semigroups (which are like monoids, but do not require a zero element) come up when we know that an aggregation will at least consume one element — this is the case when grouping elements into a map, as each sub-aggregate for a given key will have at least one element, otherwise the key simply would not be in the map.

In our Scala embedding, we represent aggregations using the standard monoid and semigroup instances⁷ of the types involved in the **yield**-expression of the query. In order to use non-standard instances (such as product on integers instead of sum), we use zero-overhead wrapper types; for example, we have `product(x:N)` of type `Product[N]`, `max(x:O)` of type `Max[O]` (for types `O` with an `Ordering` instance), etc.

⁷We use the open-source *cats* functional programming library for Scala, which provides type classes such as `Monoid` and `CommutativeMonoid`, as well as many standard instances (<https://github.com/typelevel/cats>).

Many aggregation types are semigroups but not monoids; for example, *minimum* on natural numbers or *union* on non-empty sets. In particular, we defined the `NonZero[N]` and `NonEmpty[X]` wrapper data types, which are zero-overhead “phantom subtypes” (so that `NonZero[N] <: N` and `NonEmpty[Xs] <: Xs`) that statically add more information to a type — a sort of simple type refinement — and these types are only semigroups when their wrapped type is a monoid. Note that any semigroup can be lifted to a monoid by wrapping it in an `Option` type, where `None` becomes the ad-hoc zero element, but some semigroups actually have more natural monoid generalizations than wrapping them in an `Option` type. For example, the canonical monoid form of `NonZero[Nat]` is `Nat`, and `Nat` itself is both a semigroup *and* a monoid.

Naturally, it should be illegal to write a comprehension that, for instance, aggregates the minimum age in a list of persons, i.e., `for { p <- persons } yield min(p.age)` (because if `persons` is empty, the result is ill-defined). However, it would make for a poor user experience to flat-out reject such queries and require users to write `yield Some(min(p.age))`; instead, we defined a type class which automatically lifts a semigroup to its “canonical monoid” when required. In the case above, it will give our query return type `Option[Min[Nat]]`. On the other hand, `count()` has return type `NonZero[Nat]` whose canonical monoid is `Nat`, not `Option[NonZero[Nat]]`, so a query ending with `yield count()` will have return type `Nat`, while a query ending with `yield count().groupedBy(k)` (which is really syntactic sugar for the singleton `Map(k -> 1)`) will have return type `Map[K, NonZero[Nat]]`. Table 10.1 gives some more examples.

10.6.3 Heterogeneous Collection Types

In its original formulation, the monoid comprehension calculus of Fegaras and Maier [1995, 2000] distinguishes between whether the source collection monoids are ordered, may contain repeated elements, or both — which determines which properties the result monoid should have, respectively: commutative, idempotent, or both (to have the kind of properties alluded to in the previous section). We refine and generalize these notions with more source properties and their associated monoid restrictions, namely: if the source collection is `NonEmpty` the result only needs to be a semigroup; and if the source is *not* known to be finite, then the result monoid must be what we call “lazy” or “incremental” (this allows aggregating streams and defining infinite stream pipelines).

All these conditions and restrictions are enforced statically via Scala’s type system, using implicit-based overloading together with Scala’s mechanism for prioritization of implicit search, so that the most specific (i.e., the less restrictive) `for` comprehension interface is selected automatically depending on the types of the source collections.

10.7 Optimizing Monoid Comprehension Queries with Squid

The query language we described in the previous sections is user-friendly and safe, relying on compile-time implicit resolution and type classes. But because it is directly embedded as a Scala library, queries expressed in this way can execute very slowly. This is mainly due to:

- The absence of query optimization and query planning, which are normally performed by performance-oriented query engines: users typically express queries optimizing for clarity and intuition, rather than performance, so there are often better ways of executing a given query than by naively executing each of its subqueries as they were written by the user; for example, we may want to select efficient join implementations based on the shape and properties of each particular subquery or data source, as well as unnest subqueries that are embedded inside predicates or aggregated results.
- The level of indirection and inefficiency introduced by our high-level abstractions. For example, grouping queries use lots of intermediate immutable map objects to compose their results (in particular, one singleton map is created *for each* element of the result being aggregated); moreover, we perform most monoid operations through virtual calls, sometimes through several layers of abstraction, such as when aggregating the previously-mentioned maps using composed monoid type class instances.

Both problems can be solved by deeply embedding monoid comprehension queries, as we have done for the query DSL presented in Section 9.4. Once the queries are internally represented in a way that can be inspected and transformed, we can apply the many well-known query optimization techniques which have been devised in the literature [Fegaras and Maier, 2000, Fegaras and Noor, 2018], solving the first problem; moreover, we can then emit more efficient code for the optimized queries by generating low-level imperative code that implements the high-level functional query operations, solving the second problem. In addition, the techniques described in Chapter 4 for performing compile-time optimization of user code (using the `optimize` macro), using the `@embed` annotation to see through intermediate user definitions, as well as online normalizing rewriting, also come in useful in this context.

In the rest of this section, we briefly outline our solution to optimizing the monoid comprehension calculus using Squid, without going into too much detail.

10.7.1 Motivating Example

As an example, consider the following monoid comprehension query, which lists the best-paid employees and their salary, for each department, and performs some side effect for good measure:

```
def bestPaidPerDept(ds: List[Department], es: Set[Employee])
  : List[(Department, Option[(Set[Employee], Int)])] = {
  for (d <- ds) yield {
    log("Processing department: " + d.name)
    val emp_sal_opt = for (e <- es if e.dept == d) yield Set(e).maxBy(e.salary)
    List((d, emp_sal_opt.map(_._toPair)))
  }
}
```

Above, we assume that `Set(e).maxBy(e.salary)` returns a `MaxBy[Set[Employee], Int]`, on which the method `toPair` returns a `(Set[Employee], Int)`. The `MaxBy[M, N]` data type is just a wrapper over a pair of an `M` and an `N`, with a monoid instance which accumulates all `M` corresponding with the maximum `N` — the properties of that monoid instance are inherited by the properties of the `M` monoid.⁸

A straightforward but naive execution of this query would loop over the departments `ds`, and at each iteration loop over the entire employees table `es`. But there are asymptotically more efficient ways of computing the same result; for example, taking notice of the equality predicate `e.dept == d`, we could compute a hash join of `ds` and `es`.

10.7.2 Optimization Approach

To achieve the better optimization, users of our monoid comprehension library can wrap their code in the library's `optimize` block, as follows:

```
def bestPaidPerDept ... = optimize {
  ... // query as before
}
```

Our goal is to then generate optimized lower-level code which looks like the following:

```
def bestPaidPerDept ... = {
  val result_0 = mutable.HashMap[Department, (mutable.Buffer[Employee], Int)]()
  val eit = employees.iterator
  while (eit.hasNext) {
    val e = eit.next
    val d = e.dept
    val s = e.salary
    result_0(d) = Some(result_0.get(d) match {
      case r @ Some((buff, n)) =>
        if (s == n) {
```

⁸Note that the query above would have been rejected if we had written something like `List(e).maxBy(e.salary)`, because `List` is not commutative and idempotent, unlike the `es` collection we perform the comprehension from.

```
        buff += e
        r
    } else if (s > n) (Buffer(e), s) else r
    case None => (Buffer(e), s)
  })
}
val result_1 = mutable.ListBuffer[(Department, Option[(Set[Employee], Int)])]()
val dit = employees.iterator
while (dit.hasNext) {
  val d = dit.next
  log("Processing department: " + d.name)
  val r = result_0.get(d) match {
    case Some((buff, n)) => Some((buff.toSet, n))
    case None => None
  }
  result_1 += ((d, r))
}
result_1.toList
}
```

Performance-wise, the code above is an improvement in two ways: first, it uses only low-level imperative constructs with little indirection, notably for the intermediate collections (mutable sets, map, and list buffer), avoiding the basic overhead of functional abstractions; and second, it traverses each of `Department` and `Employee` only once — the latter was traversed `ds.size` times in the naive execution.

We do *not* want users to have to write such code manually: it is much harder to read, write, debug, maintain, etc. None of the existing programming-languages-centered optimization approaches I know are able to perform these kinds of both low-level and high-level query optimizations on plain user code expressed in a general-purpose language. The one I know which comes closest is probably the work by Fegaras and Noor [2018], also a compile-time Scala embedding approach, though they use their own parser and type checker, limiting the language integration aspect and contributing to an impedance mismatch (although they do allow calling external Scala functions from the query).

10.7.3 Deeply Embedding Monoid Comprehensions

The first step in building an optimizing compiler for our DSL is to turn plain Scala code into precise internal representations of the queries the code represents. We do that just like in Section 9.4.4, by defining a `liftQuery` function to recursively compose the internal query representation:

```
def liftQuery[T: CodeType, C](q: Code[T, C]): QueryRepr[T, C] = q match { ... }
```

10.7. Optimizing Monoid Comprehension Queries with Squid

The `QueryRep` data type we use is largely similar to the one seen in Section 9.4.3, except that it focuses on the constructs of monoid comprehension instead of the more relational-algebra DSL of Section 9.4. In particular, we have a representation for general comprehensions:

```
case class Comprehension[R: CodeType, C]  
  (productions: Productions[R, C], mon: StagedMonoid[R, C])  
  extends QueryRep[R, C]  
  
sealed abstract class Productions[R, -C]  
  
case class Production[A: CodeType, R, C, VCtx]  
  (src: Path[A, C], v: Variable[A] { type Ctx = VCtx },  
   rest: Productions[R, C & VCtx])  
  extends ProductionBase[R, C]  
  
case class Yield[R, C](pred: QueryRep[Bool, C], cde: QueryRep[R, C])  
  extends Productions[R, C]
```

The `Path` data type represents the source of each production in a comprehension; it will usually be a collection, or the results of a previous comprehension.

As we traverse the query, we lift monoid instances into `StagedMonoid` representations. The goal is to recognize monoids for which we have low-level imperative implementations available; for instance, as shown in the previous subsection, to aggregate maps efficiently we use the `mutable.Map` data structure from the Scala standard library instead of the less efficient `immutable.Map`, though we still convert the end result of each query into the immutable representation mandated by our functional monoid comprehension calculus interface.

10.7.4 Query Rewriting and Planning

The query rewriting we perform follows the same patterns as seen earlier, applying well-known optimization techniques available for the monoid comprehension calculus. Following Fegaras and Noor [2018], in order to plan efficient monoid query implementations, we actually turn our staged monoid comprehension representation into a more traditional *relational* query plan representation (like the one shown in Section 9.5.2), from which we can generate the usual fast low-level query implementations.

Conclusions and Future Work

The problems of optimizing programs and of designing efficient software systems is as old as the field of computer science. A great variety of approaches have been proposed, but there is still tremendous progress to be made in the domain: we are still far from having the tools to easily reap most of the benefits of high-level languages without paying for them at runtime.

In this thesis, we have explored approaches focusing on the manipulation of program fragments in a statically-typed setting. We have extended the state of the art in several directions: by giving statically-typed quasiquotes analytic capabilities, such as pattern matching and rewriting code values, by enforcing hygiene via a new affine type system to manipulate bindings as first-class entities, and by generalizing code manipulation to classes, and not just expressions, via staged classes.

This has allowed us to define several new approaches to designing efficient systems, including a polymorphic yet efficient library for linear algebra, a stream fusion engine improving on the state of the art, a demonstration of query compilation by rewriting, and a staged SQL database system prototype.

Moreover, we have explored new techniques related to efficient data processing and program optimization, such as an embedded domain-specific language for expressing queries over collections of data, an approach to partial graph reduction approaches to optimizing functional programs aggressively in the presence of recursion, improving on existing inlining approaches.

In the future, there are many promising avenues which I would like to explore on the same topic as this thesis:

- Make use of a graph-based intermediate representation technique I have developed during my PhD⁹ to build more advanced domain-specific optimizing compilers, which could see through function definitions while avoiding code duplication. An idea would be to let users annotate existing nodes in the graph with custom representations storing arbitrary information, and use rewrite rules to combine such representations, in order to influence the generation of efficient code from the corresponding representations. The

⁹This graph-based intermediate representation work was not presented here because it is quite independent from the Squid framework, which was the focus of this thesis. However, this graph IR *can* be used as one of the possible Squid IR implementations, though it is still work in progress.

Conclusions and Future Work

graph IR would help bring together these different user-defined representations across function boundaries, allowing the optimal compilation of domain-specific programs.

- Developing further the staged classes work, perhaps by integrating it more tightly with an existing general-purpose programming language, so as to improve the metaprogramming user experience.
- Developing a more comprehensive database compilation framework, integrated with a Scala EDSL for monoid comprehension (as in Chapter 10), and with support for being integrated into user applications at compilation time, providing support for fast lightweight database features without leaving the host programming language. In fact, I have already started this work with the help of several students, under the code name *dbStage*.

A Improved GADT Reasoning in Scala

Generalized algebraic data types (GADTs) are an expressive programming language feature which lets programmers encode advanced type-based invariants as part of the definition of a program's data types. These invariants can be used to restrict the possible run-time shapes taken by such structures, as well as to existentially quantify type information inside them.

GADTs are an important tool in the design of type-safe language representations, and especially popular in the field of domain-specific language compilers. As mentioned in the previous chapter, some of the typing problems that arise in the context of pattern matching with statically-typed quasiquotes (existential types and refined types) require specifically the same reasoning power as GADTs.

However, the interaction of GADTs with subtyping has been shown to be non-trivial and potentially problematic [Scherer and Rémy, 2013]. GADTs have been notoriously difficult to implement correctly in Scala, due to Scala's advanced subtyping constructs not found in other languages with GADTs. Both major Scala compilers, Scalac and Dotty, were recently known to have type soundness holes related to GADTs. In particular, covariant GADTs have led to paradoxes related to Scala's inheritance model.

This has been a problem for Squid, since its statically-typed quasiquotes require support for GADT reasoning. To overcome the shortcomings of the Scala compiler, Squid macros have reimplemented some ad-hoc GADT reasoning to offer a safe code manipulation interface for users, who would otherwise have to resort to unsafe casts. However, the soundness of these extensions deserves some special consideration.

In this chapter, we informally explore foundations for GADTs within Scala's core type system (the pDOT calculus), in order to guide a principled understanding and implementation of GADTs in both Scala and Squid.

A.1 Introduction

Generalized algebraic data types (GADT) were proposed to encode expressive invariants through types [Xi et al., 2003, Cheney and Hinze, 2003, Kennedy and Russo, 2005]. For instance, Figure A.1 defines a GADT to represent well-typed terms of simply-typed λ -calculus, similarly to Rompf and Odersky [2010]. `Expr` is a GADT because each of its cases extends `Expr` with different type arguments. The `eval` function maps each value of type `Expr[A]` into a value of type `A`.

```
enum Expr[A] {
  case Lit(n: Int)                                extends Expr[Int]
  case Var[A](a: A)                                extends Expr[A]
  case Add(lhs: Expr[Int], rhs: Expr[Int])          extends Expr[Int]
  case Fun[A, B](fun: Expr[A] => Expr[B])           extends Expr[A => B]
  case App[A, B](fun: Expr[A => B], arg: Expr[A])    extends Expr[B]
}

def eval[A](e: Expr[A]): A = e match {
  case Lit(n)      => n
  case Var(x)      => x
  case Add(a,b)    => eval(a) + eval(b)
  case f: Fun[a,b] => (x: a) => eval(f.fun(Var(x)))
  case App(fun,arg) => eval(fun)(eval(arg))
}
```

Figure A.1 – GADT in Scala, using Dotty’s new enum syntax.

Why does type checking the `eval` function require special reasoning? First, in all but the `Var` case, the type of the scrutinee `e` is refined from `Expr[A]` to a more precise type. For example, `Lit` extends `Expr[Int]`, so if `e` matches `Lit(n)`, we can deduce that `A = Int`. This allows `n`, which has type `Int`, to agree with `eval`’s expected return type `A`. Second, in addition to refining `A`, the `Fun` and `App` cases uncover existential types (unknown types that do not appear in the function’s signature); in the `App` case, which matches against patterns of type `App[X, A] <: Expr[A]`, type `X` is unknown, but has to be treated consistently as it appears in the two extracted subexpressions `fun` and `arg`. In the `Fun` case, we have to use a type pattern `f: Fun[a, b]` to bind the uncovered existential type `a` so we can use it in a required type annotation.

Today, this specific example already works well in Dotty, the future Scala 3 compiler. However, there are several lingering unresolved issues with GADTs in Scala:

Subtle soundness issues. Scala GADTs have been plagued with type soundness issues. The `scalac` compiler uses approximate reasoning that easily leads to runtime crashes [Parreaux

et al., 2017a], while Dotty GADTs are still unsound, despite some recent substantial improvements, and are subject to ongoing work.¹

Declaration-site variance. Scala supports declaration-site variance, a convenient way of defining subtyping relationships between parameterized types. For instance, in Figure A.1 we could make `Expr` covariant to encode the λ calculus *with subtyping*. However, Dotty then rejects our definition of `eval` as ill-typed, and requires adding unsafe casts. It is unclear whether definitions like `eval` actually are unsafe, or whether Dotty is overly conservative; indeed, sound typing rules for pattern matching on open GADTs is an open problem [Giarrusso, 2013].²

To address these problems and to gain confidence in the soundness of GADTs in Scala, we believe necessary to justify them in terms of Scala’s core foundations, which have been formalized as the Dependent Object Types calculus (DOT) [Amin et al., 2016], later extended to the more comprehensive pDOT [Rapoport and Lhoták, 2019].

This short chapter makes the following contributions:

- Drawing from the `Expr` motivating example, which we believe to be quite representative, we informally sketch how to encode closed GADTs, first in full Scala, and then in a core Scala subset which can be mapped to pDOT easily. We show that `eval` actually is safe with a covariant `Expr`, and thus argue Scala should improve its support for variant GADTs (Section A.2).
- We consider the more general case of open GADTs, and sketch a minimal extension to Core Scala which allows encoding them. We explain the mismatch between such encoding and Scala’s treatment of type parameters, and following that insight we propose improvements to Scala usability which would allow sound pattern matching on variant open GADTs (Section A.3).
- We propose improvements to Scala usability related to GADTs and declaration-site variance, by giving an old experimental feature of Dotty (since removed) a second chance, providing clear motivations for it.

Our examples and our encodings are available in full at https://github.com/Blaisorblade/scala19_gadt_code.

A.2 Closed GADTs in Core Scala and DOT

In this section, we focus on encoding *closed* GADTs. By “closed,” we mean those GADTs which can be defined using the new Dotty `enum` syntax. Slightly more generally, we mean a flat class

¹See pull requests #5736 and #6398 at <https://github.com/lampepfl/dotty>.

²Scherer and Rémy [2013] did consider the problem of GADTs with subtyping, but in the context of OCaml, where they made simplifying assumptions that are not necessary in Scala, such as limiting the reasoning to type equality constraints only, as opposed to more precise subtyping constraints.

or trait hierarchy where (1) there is a single, sealed parent; (2) each implementing case is **final**; (3) each case extends the parent exactly once.

A.2.1 Encoding of ADTs and Pattern Matching

It is well-known that structurally-recursive pattern-matching on sealed hierarchies of data types can be emulated using fold functions. This technique allows encoding *algebraic data types* in System F through the Church/Böhm-Berarducci encoding [Böhm and Berarducci, 1985, Jansen, 2013]; however, recursion that is not structural becomes awkward to express and inefficient [Koopman et al., 2014].

On the other hand, in a setting with general recursion and recursive types, such as Scala (and its foundation DOT), we can instead express pattern matching through the Scott/Parigot encoding, which supports unrestricted recursion. In object-oriented languages, this encoding is equivalent to using *external* visitors [Oliveira et al., 2008, Hofer and Ostermann, 2010]: one simply defines a “visitor” method in the superclass, which will be implemented by each subclass of the hierarchy. For instance, we can encode the `Option` data type as follows:

```
abstract class Option[+A] {  
  def visit[R]: (Some[A] => R, None.type => R) => R  
}  
class Some[+A](a: A) extends Option[A] {  
  def visit[R] = (visitSome, visitNone) => visitSome(this)  
}  
object None extends Option[Nothing] {  
  def visit[R] = (visitSome, visitNone) => visitNone(this)  
}
```

A.2.2 GADTs and Object-Oriented Languages

As we have seen in Section A.1, GADTs are essentially ADTs with both existential types and type (in)equality³ proofs to be uncovered via pattern matching [Xi et al., 2003, Cheney and Hinze, 2003, Scherer and Rémy, 2013]. It is also well-understood that GADT-like type hierarchies can be defined in object-oriented languages [Kennedy and Russo, 2005, Emir et al., 2007]. The attentive reader will have guessed where we are going with this. We will encode GADTs in Scala using visitor methods. As a first step, we need to figure out how to encode existential types and subtyping proofs *without* using GADTs.

³These “inequality” proofs are *subtyping* proofs, in the case of a language with subtyping like Scala.

A.2.3 Existential types and Subtyping Proofs

In Scala, the primary way of representing existential types is via abstract type members, which are denoted using path-dependent types. However, an alternative encoding of existential types is to use higher-rank polymorphism [Böhm and Berarducci, 1985], which we will use in our first encoding approach.

On the other hand, Scala has first-class subtyping proofs thanks to its bounded abstract type members. Indeed, getting hold of an object `ev` of type `ev: { type Ev >: S <: T }` is equivalent to having a *proof*, *evidence*, or *witness* that `S <: T`.

Though the DOT calculus does not require the explicit usage of `ev` to leverage such proof (having `ev` in the typing context being considered sufficient), this property is known to make type checking in DOT undecidable [Nieto, 2017, Rompf and Amin, 2015a]. In practice, Scala users normally have to apply these proofs explicitly. For example, if one wishes to “upcast” a value `s` of type `S` to a type `T` where `S <: T` does not hold syntactically, one has to write `s: ev.Ev` (a *type ascription*). We can make this approach more convenient by defining the following data type, used for manipulating subtyping proofs, which also doubles as an implicit conversion:

```
import scala.language.implicitConversions

abstract class <:< [-A, +B] extends Conversion[A,B] {
  type Ev >: A <: B
  def apply(a: A): B = a: Ev
}

implicit def Refl[A]: A <:< A = new {
  type Ev = A
}
```

This is the same as the data type of the same name `<:<` defined in the standard library, except that we have an additional `Ev` type member, which can be leveraged when the automatically-inserted `apply` function is not enough on its own. For example, we can convert a list `ls` of type `List[S]` into a `List[T]` given some `ev: S <:< T` at no runtime cost, by writing `ls: List[ev.Ev]`.

A.2.4 Closed GADT Encoding in Scala

Figure A.2 shows the encoding of a *covariant version* of `Expr` in Scala — that is, we show how to encode pattern matching on covariant GADTs using other mechanisms. Thanks to this encoding, the soundness of closed GADTs with declaration-site variance reduces to the soundness of the rest of Scala (without GADTs).

We use implicit function types [Odersky et al., 2017], of syntax `given S => T`, in order to allow callers to implicitly leverage the subtyping evidence relevant to each case. In the `Lit` class’ implementation of `visit`, an evidence of `Int <:< A` is implicitly created and passed to the `Lit`

```

abstract class Expr[+A] {
  def visit[R]: (
    Lit: given (Int <: A) => Lit => R,
    Var: Var[A] => R
    Add: given (Int <: A) => Add => R,
    App: [B] => App[B,A] => R,
    Fun: [B, C] => given ((B => C) <: A) => Fun[B,C] => R,
  ) => R
}

final case class Lit(n: Int) extends Expr[Int] { s =>
  override def visit[R] =
    (Lit, Add, App, Fun, Var) => Lit.apply(this)
}
// other cases similarly defined...

def eval[A](e: Expr[A]): A = e.visit[A](
  Lit = l => l.n,
  Add = p => eval(p.lhs) + eval(p.rhs),
  App = [B] => a => eval(a.fun).apply(eval(a.arg)),
  Fun = [B, C] => f => ((x: B) => eval[C](f.fun(Var(x)))),
  Var = v => v.a
)

```

Figure A.2 – An encoding of the closed, covariant GADT in Figure A.1. This code leverages polymorphic function types, which use the $[X] \Rightarrow F[X]$ syntax, analogous to system F's Λ/\forall binders.

visitor (because within class `Lit`, we know that `A` is a supertype of `Int`). This piece of subtyping evidence is then leveraged, in the `Lit` case of `eval`, to upcast `l.n`, which has type `Int`, into type `A`.

While the encoding is elegant and intuitive, it is not fully satisfactory since full Scala is known to still have soundness holes (see e.g., [Amin and Tate, 2016]). Therefore, we now propose an encoding into *Core Scala*, a Scala subset that we describe next.

A.2.5 Core Scala and DOT

We define *Core Scala* as the specific subset of Scala that can be translated into the dependent object type calculus in a straightforward manner. Since we will make use of singleton types, which are not supported in DOT, we target in particular the pDOT dialect of DOT [Rapoport and Lhoták, 2019], which soundly extends DOT with singleton types (and with paths, a feature we do not use).

DOT and pDOT do not directly support classes, but there are several examples in the literature of how to encode them [Amin et al., 2016, Gruetter, 2016, Rapoport and Lhoták, 2019]. Essentially, a class is represented as an abstract type whose upper bound specifies the class API, along with some constructors for building instances of the class.

The encoding, as described above, of non-generic classes (classes without type parameters) is straightforward. Furthermore, we can encode *generic* classes as *non-generic* classes plus abstract type members: each class type parameter is turned into an abstract type member of the class, and applications of the class' type constructor to some arguments are represented as refinements of the class type (with type intersections).

For instance, a `class Foo[A] { val a: A }` can be represented as a `class FooBase { type A; val a: A }` together with a type shorthand `type Foo[A0] = FooBase { type A = A0 }`, which refines `FooBase` with type member `A = A0`. We refer to the literature for more concrete examples [Amin et al., 2016, Rapoport and Lhoták, 2019, Odersky et al., 2016].

A.2.6 Closed GADT Encoding in Core Scala

Figure A.3 shows an encoding of covariant `Expr` in Core Scala. The parameterized type alias `Expr[+A0]` is not definable in DOT but can be inlined at its call sites before translation.

Indeed, we have encoded by hand the full Figure A.3 in pDOT syntax following the class encoding mentioned earlier.⁴ This encoding can be seen in Appendix B.

One central insight of this encoding is that we do not need separate `<:<` witnesses, nor do we need polymorphic function types. This is because (1) we now use an abstract type `A` to

⁴We have not mechanically verified that this code can be typechecked in (p)DOT, due to the lack of an actual implementation of these systems.

```
type Expr[+A0] = ExprBase { type A <: A0 }

abstract class ExprBase { s =>
  type A
  def visit[R]: (
    Lit: Lit & s.type => R,
    Add: Add & s.type => R,
    App: App & s.type => R,
    Fun: Fun & s.type => R,
    Var: Var & s.type => R
  ) => R
}

abstract class Fun extends ExprBase {
  type B; type C
  type A = B => C
  val fun: Expr[B] => Expr[C]
  override def visit[R] =
    (Lit, Add, App, Fun, Var) => Fun(this)
}
// other cases similarly defined...

def eval[A](e: Expr[A]): A = e.visit[A](
  Lit = l => l.n: l.A,
  Add = p => (eval(p.lhs) + eval(p.rhs)): p.A,
  App = a => eval(a.fun).apply(eval(a.arg)): a.A,
  Fun = f => ((x: f.B) => eval[f.C](f.fun(Var(x)))): f.A,
  Var = v => v.a: v.A
)
```

Figure A.3 – An encoding of the closed GADT in Figure A.1 in Core Scala.

represent the type parameter of the same name, and A is now visible from the outside; (2) A is *refined* in each subclass of `Expr` (see the definition of `type A` in class `Fun`); and (3) in the visitor method, we intersect the types of the extracted objects with the self-type s of the current instance. Thus, following normal DOT rules for type intersections, we are able to define `eval` by simply using the abstract type A itself as subtyping proof. Interestingly, to represent GADTs in Core Scala, we did not need to add any of the typical mechanisms commonly used to type check GADTs, such as special type equality proofs and coercions [Sulzmann et al., 2011, Garrigue and Rémy, 2013].

A.2.7 Summary

We argue that Scala should handle closed GADTs well, irrelevant of variance, given the insight of how they should be represented in Scala’s core type system.

Naturally, one should develop a general formal explanation of this process — here we have merely tried to give an intuition about what that process could be. As explained further in Section A.4, this formalization effort has already been started by Waśko [2020], following the model presented here.

A.3 Open GADTs

We now show an encoding of “open” GADTs — GADTs that are not sealed or do not have a flat hierarchy. These are useful because they can be extended with new constructors in a modular way, providing a solution to the expression problem [Rompf and Odersky, 2010].

Our previous encoding does not readily generalize to open GADTs, as the `visit` method received handlers for a fixed list of constructors, while open GADTs would require different lists of constructors for different extensions.

A.3.1 Class Instance Matching

To achieve our new encoding, we assume Core Scala and (p)DOT are extended with a primitive runtime-class instance matching mechanism, which mirrors type matches in Scala:

```
s match {
  case x_1: C_1 => t1
  case x_2: C_2 => t2
  ...
}
```

This construct branches on the runtime class of s , comparing it with classes C_1 , C_2 , etc. and evaluating the corresponding branch t_1 , t_2 , etc.

Appendix A. Improved GADT Reasoning in Scala

To keep the extension simple, we only allow matching against simple class names, not arbitrary types, avoiding the complexities of Scala’s type matching syntax. Since Core Scala has no class type parameters, we also avoid soundness problems due to the erasure of type parameters (in Scala, a type match like `case _: Option[Int]` is rejected because it cannot be checked at runtime).

In order to type the RHS of each branch of the class-matching expression, we bind the corresponding pattern variable `x_i` with type `C_i & s.type` (similarly to Figure A.3). In terms of operational semantics, we must tag class instances with their class at run time, as is done in Java runtime systems. Formalizing this extension is out of scope for this chapter.

With this extension, encoding *an open version* of `Expr` becomes as simple as dropping the `visit` method from the previous encoding of Figure A.3, using type matches instead:

```
def eval[A](e: Expr[A]): A = e match {  
  case l: Lit => l.n: l.A  
  /* other cases... */  
}
```

A.3.2 Understanding an Old Paradox

Giarrusso [2013] first noticed that certain desirable typing rules on covariant open GADTs are in fact unsound. For example, given:

```
trait Expr[+A]  
class Const[+A] extends Expr[A]
```

the Scala 2 type checker would assume that if `e: Expr[A]` and if `e` is an instance of `Const`, then `e: Const[A]`. But this assumption is false, because in Scala, one can extend covariant types like `Expr` multiple times, with different type arguments. Concretely, this allows us to break the Scala 2 compiler’s assumption by, for instance, defining the object `Unsound` as follows, which extends `Expr[Int]` and is an instance of `Const`, but not of `Const[Int]`:

```
object Unsound extends Const[Any] with Expr[Int]
```

Viewing type parameters as type members not only makes the problem obvious, but also suggests how to make the assumption true without allowing definitions such as `Unsound`. In our interpretation, the classes from the above paradox instance would be translated as follows:

```
type Expr[+A] = ExprBase { type A$0 <: A }  
type Const[+A] = ConstBase { type A$1 <: A }  
trait ExprBase { type A$0 }  
class ConstBase extends ExprBase {  
  type A$0 <: A$1  
  type A$1  
}
```

Notice that while translating the type parameters of each class, it is crucial to pick *different* type member names, so they do not conflict with each other (this reflects Scala’s established semantics that the type parameters in different classes are distinct, even if they have the same name). The subtyping relationship arising from inheritance of a *variant* base class is expressed via refinements of these abstract type members.

With the above definition, the assumption from the paradox is obviously false: given $e: \text{Expr}[A]$ and $e: \text{ConstBase}$, we cannot conclude that $e: \text{Const}[A]$. Moreover, the assumption actually becomes true if we use a *different interpretation of inheritance from variant base classes*: if we instead declared **type** $A\$0 = A\1 in `ConstBase`, this would prevent further extending `Expr` with incompatible type arguments, and would allow us to derive the required type equality proofs.

A.3.3 Solution: Invariant Inheritance

Giarrusso [2013] also proposes a solution to the paradox: a syntax for “invariant inheritance,” **class** `Const[+T] extends Expr[=T]`, which forbids definitions such as `Unsound` by simply forbidding further instantiation of `Expr` in children of `Const`.

Following our new insights into the paradox, we propose this syntax to instead behave consistently with our encoding of type parameters as type members, translating those type argument marked with `=` to type equalities rather than subtype refinements.

This interpretation still forbids definitions like `Unsound` while also allowing further extensions of `Expr`, as long as they conform in their type arguments. For example, in the code below, `Z` should be able to extend `Expr` via both `Valued` and `Const`, as they are compatible:

```
trait Valued extends Expr[Int] { def v: Int }
object Z extends Const[0] with Valued { def v = 0 }
```

Remark that if we had written `Valued extends Expr[=Int]` above, the code of `Z` would have not compiled, as it would have had conflicting definitions for the parameter to `Expr`.

Interestingly, Dotty already has partial support for invariant inheritance (implemented specifically to counter Giarrusso’s paradox in common cases), but it is restricted to case classes and not expressible otherwise. This makes the approach somewhat irregular and “magical” as it is not expressible in terms of regular features, unlike all other case class features.

A.3.4 Type Parameters as Members

In Scala, class type parameters (unlike type members, as in our encoding), cannot be referenced from the outside of a class. However, there are cases where having access to these types is actually very useful.

Therefore, we propose to allow the use of ‘**type**’ as a prefix for type parameters which, in

Appendix A. Improved GADT Reasoning in Scala

analogy with ‘`val`’, would make the corresponding type publicly visible from the outside.

One possible motivating example is the following, which cannot be expressed in current Scala, and which describes a pair of two covariant expressions which happen to precisely share the *same* type argument $A = T$:

```
enum Expr[type +A] {  
  case StrLit() extends Expr[String]  
  case IntLit() extends Expr[Int]  
}  
case class ExprPair[T](  
  _1: Expr[Any] { type A = T },  
  _2: Expr[Any] { type A = T }  
)
```

The **type** modifier signifies that the type member from our encoding can be referenced (and therefore, also refined) in Scala source code outside of the class.

Based on this, we define `ExprPair` class which can only contain two `Expr` values of the same class. If `Expr` was invariant, a simple pair `(Expr[T], Expr[T])` would have this property. For variant GADTs however, this can only be expressed by refining the type members with proper type equalities. Assuming that the compiler supports proper GADT reasoning, it should then become possible to verify that the following pattern matching expression is exhaustive:

```
def m[A](p: ExprPair[A]) = p match {  
  case (StrLit, StrLit) => ... // here we know String <: A  
  case (IntLit, IntLit) => ... // here we know Int <: A  
}
```

Such a pattern match would not be exhaustive if the input was merely of type `(Expr[T], Expr[T])`. Indeed, we could for example pass in the value `(IntLit(), StrLit())` by instantiating the method `m` as `m[Any]`.

Additionally, type parameters as members would solve current limitations sometimes encountered in variant classes. Consider the following types:⁵

```
abstract class Type[A]  
abstract class Expr[+A](ty: Type[_ <: A])
```

We cannot make `ty` have type `Type[A]` here, as that would be a violation of covariance. This results in problems when one wants to extract the *precise* type representation of an `Expr`, to which we cannot give an accurate type. One can always use the encoding of Figure A.3, but it is clunky and splits type in two parts, also resulting in subpar error messages (in which the type alias will usually be expanded). Under our proposal, we could simply write:

⁵Unlike expressions, type representations like `Type[A]` generally need to be invariant, for soundness reasons.

```
abstract class Expr[type +A](ty: Type[this.A])
```

and given an `e` of type `Expr[T]`, value `e.ty` would then be typed as `Type[e.A]`, which would not violate the covariance of the type parameter `A`.

A.4 Further Work on GADTs in Scala

Squid and its GADT-reasoning extension were designed for Scala 2 around 2017–18. Since then, the GADT situation in Scala has been steadily improving.

Independently of this work, Aleksander Boruch-Gruszecki and Martin Odersky have implemented advanced GADT support in Dotty, the compiler for the upcoming version 3 of the Scala language. Their implementation records constraints based on the types of matched values, and leverages the information gathered from these constraints automatically so that users often do not have to use explicit type ascriptions. In the light of the work presented here, their approach can be likened to an elaboration phase, which automatically inserts type ascriptions in order to make the Scala code well-typed according to Scala’s core typing rules.

The work presented in this chapter was recently extended by Waśko [2020] for a master project with Aleksander Boruch-Gruszecki. Waśko made more formal the correspondence between our interpretation of GADTs in Scala and traditional GADTs, by showing how to encode the $\lambda_{2,G\mu}$ calculus of guarded recursive datatype constructors (another name for GADTs) by Xi et al. [2003] in pDOT using the technique described here.

A.5 Conclusion

GADTs in Scala have historically been poorly understood. In this chapter, we showed that they can be explained in terms of simpler features already present in Scala’s core type system. We sketched different encodings of GADTs, demonstrating the tight correspondence between, on one hand, the type (in)equality proofs and existential types that underlie traditional approaches to GADT reasoning and, on the other hand, bounded abstract type members and intersection types, which are core to Scala.

B Complete Encoding of GADT in pDOT

In this appendix, we give a complete encoding of Figure A.3 in pDOT.

```
val exprsMod =
  new { exprs =>
    type ExprBase =  $\mu$  { s =>
      type A
      val `match`:
         $\forall$  (r: { type R })
         $\forall$  (Lit: exprs.Lit & s.type => r.R,
          Plus: exprs.Plus & s.type => r.R,
          App: exprs.App & s.type => r.R,
          Fun: exprs.Fun & s.type => r.R,
          Var: exprs.Var & s.type => r.R): r.R
    }

    type Lit = exprs.ExprBase  $\wedge$  {
      type A = Int
      val n: Int
    }

    val newLit:  $\forall$ (x: Int) exprs.Lit =
       $\lambda$  x. new {
        type A = Int
        val n = x
        val `match` =
           $\lambda$  r Lit Plus App Fun Var. Lit(self)
      }

    type Plus = exprs.ExprBase  $\wedge$  {
      type A = Int
      val lhs: exprs.ExprBase  $\wedge$  { type A <: Int }
      val rhs: exprs.ExprBase  $\wedge$  { type A <: Int }
```

```

}
val newPlus:
  ∀ (lhs: exprs.ExprBase ∧ { type A <: Int },
    rhs: exprs.ExprBase ∧ { type A <: Int })
    exprs.Plus =
  λ lhs lrhs . new { self =>
    type A = Int
    val lhs = lhs
    val rhs = lrhs
    val `match` =
      λ r Lit Plus App Fun Var. Plus(self)
  }

type App = exprs.ExprBase ∧ μ { self =>
  type B
  val fun:
    exprs.ExprBase { type A <: self.B } =>
    exprs.ExprBase { type A <: self.A }
  val arg: exprs.ExprBase { type A <: self.B }
}
val newApp: ∀(bT: { type B }, aT: { type A })
  ∀ (fun: exprs.ExprBase { type A <: bT.B } =>
    exprs.ExprBase { type A <: aT.A })
  ∀ (arg: exprs.ExprBase { type A <: bT.B }).
    exprs.App ∧ {
      type A = aT.A; type B = bT.B } =
  λ bT aT lfun larg. new {
    type A = aT.A
    type B = bT.B
    val fun = lfun
    val arg = larg
    val `match` =
      λ r Lit Plus App Fun Var. App(self)
  }

type Fun = exprs.ExprBase ∧ μ { self =>
  type B
  type C
  type A = self.B => self.C
  val fun:
    exprs.ExprBase { type A <: self.B } =>
    exprs.ExprBase { type A <: self.C }
}

```

```

val newFun:  $\forall$ (bT: { type B }, cT: { type C })
   $\forall$  (fun: exprs.ExprBase { type A <: bT.B } =>
    exprs.ExprBase { type A <: cT.C }).
    exprs.Fun  $\wedge$  {
      type B = bT.B; type C = cT.C } =
   $\lambda$  bT cT lfun. new { self =>
    type B = bT.B
    type C = cT.C
    type A = self.B => self.C
    val fun = lfun
    val `match` =
       $\lambda$  r Lit Plus App Fun Var. Fun(self)
  }

type Var = exprs.ExprBase  $\wedge$   $\mu$  { self =>
  val a: self.A
}
val newVar:
   $\forall$  (aT: {type A})  $\forall$ (a: aT.A)
    exprs.Var  $\wedge$  { type A = aT.A } =
   $\lambda$  aT la. new {
    type A = aT.A
    val a = la
    val `match` =
       $\lambda$  r Lit Plus App Fun Var. Var(self)
  }
} :
 $\mu$  { exprs =>
  /* Omitted, see discussion below. */
}

// encode function eval as function eval.rec
val eval = new { self =>
  val rec:  $\forall$ (e: exprsMod.ExprBase) e.A =  $\lambda$ e.
    e.`match`
    ( $\lambda$  l. l.n)
    ( $\lambda$  p. eval.rec(p.lhs) + eval.rec(p.rhs))
    ( $\lambda$  a. eval.rec(a.fun)(eval.rec(a.arg)))
    ( $\lambda$  f.  $\lambda$ x. eval.rec(f.fun(exprsMod.newVar(x))))
    ( $\lambda$  v. v.a)
}

```

Value `exprsMod` encodes the whole module containing the code in Figure A.3. Each class `Foo`

Appendix B. Complete Encoding of GADT in pDOT

is encoded by declaring a type member `Foo` and a value member `newFoo`, which encodes the constructor of `Foo`.

To improve readability, we omit type annotations from terms when they can be inferred directly from their types. We also omit the type of `exprsMod`: this type declares the interface for `exprsMod` members. For a value member definition `val foo: FooType = ...`, the interface is just `val foo: FooType`. For type members `type Foo = Body` the interface we choose is `type Foo <: Body`. This omits the lower bound, preventing the creation of values of `Foo` from outside `exprsMod`, except by going through the encoded constructor `newFoo`; this is how nominal class types are encoded by (p)DOT's structural type system [Amin et al., 2016]. For instance, `Lit` is encoded as follows:

```
type Lit <: exprs.ExprBase ^ {  
  type A = Int  
  val n: Int  
}  
  
// Value members are just declared, giving their types.  
val newLit: ∀(x: Int) exprs.Lit
```

C Organization of the Streams Optimizer

In this appendix, we give more details on the way a library should be structured in order to benefit from Squid-based Quoted Staged Rewriting, using the streams library seen in Chapter 5 as an example.

Scala Restrictions. Squid makes extensive use of macros, which currently have some restrictions: Scala code defined in some project P cannot be executed at compile-time in P itself or in a project that P depends on — one may have to “stratify” program definitions into different sub-projects.

The Strm Library. It is organized in a single project as follows: in the `lib` package, the shallow Strm definitions as seen in Figure 5.1, with an `@embed` annotation to automatically lift method definitions; in the `compiler` package, the ANF-based “embedding” used as the IR in which to manipulate the code (Squid supports different IRs [Parreaux et al., 2017b]), defined as follows:

```
object Embedding extends SchedulingANF
  with OnlineOptimizer with StandardEffects
{ object Desug extends Transformer with Desugaring
  object Norm extends Transformer
    with StandardNormalizer with LogicNormalizer
    def pipeline = Desug.pipeline andThen Norm.pipeline
    embed(Strm)
  }
```

Object Embedding extends the SchedulingANF base IR and the StandardEffects trait to benefit from effect annotations on standard Scala constructs. It extends OnlineOptimizer in order to perform some online rewriting — provided via the pipeline method, which applies some desugaring and then some normalization. The `embed(Strm)` call, executed when Embedding is initialized, registers in this IR the Strm methods lifted earlier by `@embed`.

The stream fusion optimizer itself is implemented in the `compiler.StrmOptimizer` class, which defines successive optimization phases Flow, Lowering and LowLevel:

```
class StrmOptimizer extends Optimizer
{ def pipeline = (
  Flow.pipeline
  andThen Lowering.pipeline
```

Appendix C. Organization of the Streams Optimizer

```
    andThen LowLevel.pipeline )
  }
  object Flow extends Embedding.Transformer
    with SimpleRuleBasedTransformer
    with BottomUpTransformer
    with FixPointTransformer
  { rewrite { ... } }
  object Lowering extends Embedding.Transformer with ...
  object LowLevel extends Embedding.Transformer with ...
```

The way Flow is defined above is equivalent to the annotation-based way seen in the thesis, which is only syntax sugar (i.e., @bottomUp @fixedPoint **val** Flow = **rewrite**{ ... }). The role of LowLevel is to apply low-level transformations at the end of the pipeline, such as flattening variables holding an option type into a boolean variable isDefined and an unwrapped currentValue variable.

In order to use this optimizer, one has to instantiate class StaticOptimizer[strm.compiler.StrmOptimizer] and, *from another project*, invoke its optimize{ ... } macro.

D Code of the Microbenchmarks

Below we list the code corresponding to the Baseline (name postfixed with `_baseline`) and Default (name postfixed with `_shallow`) microbenchmarks in Figure 5.4. The code for Rewritten is the same as the code for Default but surrounded with an `optimize{...}` block. The code of Staged is the one presented in [Kiselyov et al., 2017]. Most of the Baseline code below was reused from [Kiselyov et al., 2017] with the author’s authorization.

The inputs consisted of arrays of small integer elements with cardinalities $|v| \models 100,000,000$, $|vHi| \models 10,000,000$, $|vLo| \models 10$, $|vFaZ| \models 10,000$, and $|vZaF| \models 10,000,000$.

```
def sum_baseline(): Int = {
  var i=0
  var sum=0
  while (i < v.length) {
    sum += v(i)
    i += 1
  }
  sum
}
def sum_shallow(): Int = fromArray(v).fold(0)(_ + _)
def sumOfSquares_baseline(): Int = {
  var i=0
  var sum=0
  while (i < v.length) {
    sum += v(i) * v(i)
    i += 1
  }
  sum
}
def sumOfSquares_shallow(): Int = {
  fromArray(v).map(x => x * x).fold(0)(_ + _)
}
```

Appendix D. Code of the Microbenchmarks

```
def sumOfSquaresEven_baseline(): Int = {
  var i=0
  var sum=0
  while (i < v.length) {
    if (v(i) % 2 == 0)
      sum += v(i) * v(i)
    i += 1
  }
  sum
}

def sumOfSquaresEven_shallow(): Int = {
  fromArray(v)
  .filter(y => y % 2 == 0)
  .map(x => x*x)
  .fold(0)(_ + _)
}

def cart_baseline(): Int = {
  var d, dp=0
  var sum=0
  while (d < vHi.length) {
    dp = 0
    while (dp < vLo.length) {
      sum += vHi(d) * vLo(dp)
      dp +=1
    }
    d += 1
  }
  sum
}

def cart_shallow(): Int = {
  fromArray(vHi)
  .flatMap(d => fromArray(vLo).map (dp => dp * d))
  .fold(0)(_ + _)
}

def filters_baseline(): Int = {
  var i=0
  var sum=0
  while (i < v.length) {
    if (v(i) > 1 && v(i) > 2 && v(i) > 3 && v(i) > 4
        && v(i) > 5 && v(i) > 6 && v(i) > 7)
      sum += v(i)
    i += 1
  }
}
```

```

    sum
}
def filters_shallow(): Int = {
  fromArray(v)
    .filter(x => x > 1)
    .filter(x => x > 2)
    .filter(x => x > 3)
    .filter(x => x > 4)
    .filter(x => x > 5)
    .filter(x => x > 6)
    .filter(x => x > 7)
    .fold(0)(_ + _)
}
def maps_baseline(): Int = {
  var i=0
  var sum=0
  while (i < v.length) {
    sum += v(i) * 1*2*3*4*5*6*7
    i += 1
  }
  sum
}
def maps_shallow(): Int = {
  fromArray(v)
    .map(x => x * 1)
    .map(x => x * 2)
    .map(x => x * 3)
    .map(x => x * 4)
    .map(x => x * 5)
    .map(x => x * 6)
    .map(x => x * 7)
    .fold(0)(_ + _)
}
def dotProduct_baseline(): Int = {
  var counter = 0
  var sum = 0
  while (counter < vHi.length) {
    sum += vHi(counter) * vHi(counter)
    counter += 1
  }
  sum
}
def dotProduct_shallow(): Int = {

```

Appendix D. Code of the Microbenchmarks

```
    fromArray(vHi)
    .zipWith(fromArray(vHi))(((a,b) => a * b))
    .fold(0)(_ + _)
  }
def flatMap_after_zipWith_baseline(): Int = {
  var counter1 = 0
  var sum = 0
  while (counter1 < vFaZ.length) {
    val item1 = vFaZ(counter1) + vFaZ(counter1)
    var counter2 = 0
    while (counter2 < vFaZ.length) {
      val item2 = vFaZ(counter2)
      sum += item2 + item1
      counter2 += 1
    }
    counter1 += 1
  }
  sum
}
def flatMap_after_zipWith_shallow(): Int = {
  val xs = vFaZ
  val ys = vFaZ
  fromArray(xs)
  .zipWith(fromArray(xs))(_ + _)
  .flatMap(x => fromArray(ys).map(y => (x + y)))
  .fold(0)(_ + _)
}
def zipWith_after_flatMap_baseline(): Int = {
  var sum = 0
  var index1 = 0
  var index2 = 0
  var flag1 = (index1 <= vZaF.length - 1)
  while (flag1 && (index2 <= vZaF.length - 1)) {
    var el2 = vZaF(index2)
    index2 += 1
    var index_zip = 0
    while (flag1 && (index_zip <= vZaF.length - 1)) {
      var el1 = vZaF(index_zip)
      index_zip += 1
      var elz = vZaF(index1)
      index1 += 1
      flag1 = (index1 <= vZaF.length - 1);
      sum = sum + elz + el1 + el2
    }
  }
}
```

```

    }
  }
  sum
}

def zipWith_after_flatMap_shallow(): Int = {
  val xs = vZaF
  val ys = vZaF
  fromArray(xs)
  .flatMap(x => fromArray(ys).map(y => (x + y)))
  .zipWith(fromArray(xs))(_ + _)
  .fold(0)(_ + _)
}

def flatMap_take_baseline(): Int = {
  var counter1 = 0
  var counter2 = 0
  var sum = 0
  var n = 0
  var flag = true
  val size1 = v.length
  val size2 = vLo.length
  while (counter1 < size1 && flag) {
    val item1 = v(counter1)
    while (counter2 < size2 && flag) {
      val item2 = vLo(counter2)
      sum = sum + item1 * item2
      counter2 += 1
      n += 1
      if (n == 20000000)
        flag = false
    }
    counter2 = 0
    counter1 += 1
  }
  sum
}

def flatMap_take_shallow(): Int = {
  val vHi = v
  fromArray(vHi)
  .flatMap(x => fromArray(vLo).map(y => (x * y)))
  .take(20000000)
  .fold(0)(_ + _)
}

def zip_flat_flat_baseline(): Int = {

```

Appendix D. Code of the Microbenchmarks

```
val vHi = v
var sum = 0
var index11 = 0
var index12 = 0
var index21 = 0
var index22 = 0
var taken = 0
var goOn = true
val toTake = 20000000
while (index11 < vHi.length && taken < toTake && goOn) {
  index12 = 0
  while (index12 < vLo.length && taken < toTake && goOn) {
    val e11 = vHi(index11) * vLo(index12)
    if (index22 > vHi.length) {
      index21 += 1
      index22 = 0
    }
    if (index21 >= vLo.length) goOn = false
    else if (index22 < vHi.length) {
      sum += e11 + vLo(index21) * vHi(index22)
      taken += 1
      index22 += 1
    }
    index12 += 1
  }
  index11 += 1
}
sum
}

def zip_flat_flat_shallow(): Int = {
  val s0 = fromArray(v)
    .flatMap(x => fromArray(vLo).map(y => (x * y)))
  val s1 = fromArray(vLo)
    .flatMap(x => fromArray(v).map(y => (x * y)))
  (s0 zipWith s1)(_ + _).take(20000000).fold(0)(_ + _)
}

def zip_filter_filter_baseline(): Int = {
  val xs = vHi
  val ys = vHi
  var counter1 = 0
  var counter2 = 0
  var sum = 0
  val size1 = xs.length
```

```
val size2 = ys.length
while (counter1 < size1 && counter2 < size2) {
  while (!(xs(counter1) > 5) && counter1 < size1)
    counter1 += 1
  if (counter1 < size1) {
    val item2 = ys(counter2)
    if (item2 > 5) {
      sum += xs(counter1) + item2
      counter1 += 1
    }
    counter2 += 1
  }
}
sum
}

def zip_filter_filter_shallow(): Int = {
  val xs = vHi
  val ys = vHi
  ( fromArray(xs) filter (_ > 5)
    zipWith
    fromArray(ys) filter (_ > 5)
  )(_ + _).fold(0)(_ + _)
}
```


Bibliography

- Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.
- H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams, D. P. Friedman, E. Kohlbecker, G. L. Steele, D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, M. Wand, William Clinger, and Jonathan Rees. Revised⁴ report on the algorithmic language Scheme. *SIGPLAN Lisp Pointers*, IV(3):1–55, July 1991. ISSN 1045-3563. doi: 10.1145/382130.382133. URL <https://doi.org/10.1145/382130.382133>.
- David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. ISBN 0321227255.
- Michael D. Adams. Towards the essence of hygiene. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 457–469, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333009. doi: 10.1145/2676726.2677013. URL <https://doi.org/10.1145/2676726.2677013>.
- Yanif Ahmad and Christoph Koch. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*, 2(2):1566–1569, 2009.
- Nada Amin and Ross Tate. Java and Scala’s Type Systems Are Unsound: The Existential Crisis of Null Pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 838–848, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984004. URL <http://doi.acm.org/10.1145/2983990.2984004>.
- Nada Amin, Karl Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The Essence of Dependent Object Types. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *WadlerFest 2016*. Springer, 2016.
- Johan Anker and Josef Svenningsson. An EDSL approach to high performance Haskell programming. In *ACM Haskell Symposium*, pages 1–12, 2013.

Bibliography

- Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-41695-7.
- Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD’15, pages 1383–1394, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9.
- Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *8th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 169–178. IEEE, 2010.
- Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The Design and Implementation of Feldspar an Embedded Language for Digital Signal Processing. In *Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages*, IFL’10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24275-5.
- Alan Bawden. Quasiquotation in Lisp. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 4–12. ACM, 1999.
- Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *arXiv preprint arXiv:1502.05767*, 2015a.
- Atilim Gunes Baydin, Barak A Pearlmutter, and Jeffrey Mark Siskind. DiffSharp: Automatic Differentiation Library. *arXiv preprint arXiv:1511.07727*, 2015b.
- James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.
- Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Binders by day, labels by night: Effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi: 10.1145/3371116. URL <https://doi.org/10.1145/3371116>.
- Christian Bischof, Peyvand Khademi, Andrew Mauer, and Alan Carle. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering*, 3(3): 18–32, 1996.
- Christian H Bischof, HM Bucker, Bruno Lang, Arno Rasch, and Andre Vehreschild. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*, pages 65–72. IEEE, 2002.

- Maximilian Bolingbroke and Simon Peyton Jones. Supercompilation by evaluation. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. doi: 10.1145/1863523.1863540. URL <http://doi.acm.org/10.1145/1863523.1863540>.
- Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effekt: Capability-passing style for type-and effect-safe, extensible effect handlers in scala. *Journal of Functional Programming*, 30, 2020.
- Eugene Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2064-1.
- Eugene Burmako. Unification of compile-time and runtime metaprogramming in scala. page 240, 2017a. doi: 10.5075/epfl-thesis-7159. URL <http://infoscience.epfl.ch/record/226166>.
- Eugene Burmako. Scala meta. <http://scalameta.org/>, 2017b. Accessed: 2017-07-20.
- Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed Lambda-programs on term algebras. *Theoretical Computer Science*, 39:135–154, January 1985. ISSN 0304-3975. doi: 10.1016/0304-3975(85)90135-5. URL <http://www.sciencedirect.com/science/article/pii/0304397585901355>.
- Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering*, pages 57–76, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-39815-8.
- Jacques Carette and Oleg Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Sci. Comput. Program.*, 76(5):349–375, May 2011. ISSN 0167-6423.
- Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.
- Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.
- Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of system R. *Commun. ACM*, 24(10):632–646, 1981.

Bibliography

- P. P. Chang and W.-W. Hwu. Inline function expansion for compiling C programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 246–257, New York, NY, USA, 1989. ACM. ISBN 0-89791-306-X.
- Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, 2012.
- Arun Chauhan and Ken Kennedy. Optimizing strategies for telescoping languages: Procedure strength reduction and procedure vectorization. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, page 92–101, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 158113410X. doi: 10.1145/377792.377812. URL <https://doi.org/10.1145/377792.377812>.
- Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. *Journal of Functional Programming*, 15(6):797–835, 2005. doi: 10.1017/S0956796805005617.
- James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.
- James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 403–416, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0.
- Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, page 143–156, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939197. doi: 10.1145/1411204.1411226. URL <https://doi.org/10.1145/1411204.1411226>.
- Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *ACM Sigplan Notices*, volume 45, pages 122–133. ACM, 2010.
- Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2): 56–68, 1940. doi: 10.2307/2266170.
- Koen Claessen, Mary Sheeran, and Bo Joel Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming*, DAMP '12, pages 21–30, NY, USA, 2012. ACM.
- Cliff Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 246–257, New York, NY, USA, 1995. ACM. ISBN 0-89791-697-2.
- Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *TOPLAS*, 17(2):181–196, March 1995. ISSN 0164-0925. doi: 10.1145/201059.201061. URL <http://doi.acm.org/10.1145/201059.201061>.

- Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pages 35–49, New York, NY, USA, 1995. ACM. ISBN 0-89791-754-5. doi: 10.1145/202529.202534. URL <http://doi.acm.org/10.1145/202529.202534>.
- Albert Cohen, Sébastien Donadio, Maria-Jesus Garzaran, Christoph Herrmann, Oleg Kise-lyov, and David Padua. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming*, 62(1): 25 – 46, 2006. ISSN 0167-6423. doi: <https://doi.org/10.1016/j.scico.2005.10.013>. URL <http://www.sciencedirect.com/science/article/pii/S0167642306000724>. Special Issue on the First MetaOCaml Workshop 2004.
- Duncan Coutts. *Stream fusion : practical shortcut fusion for coinductive sequence types*. PhD thesis, University of Oxford, UK, 2011.
- Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, page 315–326, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938152. doi: 10.1145/1291151.1291199. URL <https://doi.org/10.1145/1291151.1291199>.
- Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. Tupleware: “big” data, big analytics, small clusters. In *7th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.
- Ryan Culpepper and Matthias Felleisen. Taming macros. In *Third International Conference on Generative Programming and Component Engineering (GPCE) 2004, Vancouver, Canada, October 24-28, 2004. Proceedings*, pages 225–243, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 151–160, New York, NY, USA, 1990. ACM. ISBN 0-89791-368-X.
- Rowan Davies. A temporal-logic approach to binding-time analysis. In *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 184–195. IEEE, 1996.
- Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM (JACM)*, 48(3):555–604, 2001.
- Oege de Moor and Ganesh Sittampalam. *Generic Program Transformation*, pages 116–149. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-48506-3. doi: 10.1007/10704973_3. URL https://doi.org/10.1007/10704973_3.
- Oege de Moor and Ganesh Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269(1-2):135–162, 2001.

Bibliography

- Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI'07*, pages 2468–2473, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.
- Rina Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Learning in graphical models*, pages 75–104. Springer, 1998.
- Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11*, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307710. doi: 10.1145/2063384.2063396. URL <https://doi.org/10.1145/2063384.2063396>.
- Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Notices*, volume 48, pages 105–116. ACM, 2013.
- Zachary DeVito, Daniel Ritchie, Matt Fisher, Alex Aiken, and Pat Hanrahan. First-class runtime generation of high-performance types using exotypes. In *Proceedings of the 35th Conference on Programming Language Design and Implementation*, page 11. ACM, 2014.
- Stephen Dolan. Fun with Semirings: A Functional Pearl on the Abuse of Linear Algebra. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP '13*, pages 101–110, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0.
- Conal M. Elliott. Beautiful differentiation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP '09*, pages 191–202, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7.
- Burak Emir, Martin Odersky, and John Williams. Matching Objects with Patterns. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, Lecture Notes in Computer Science, pages 273–298. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73589-2.
- Martin Erwig and Steve Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *Journal of Functional Programming*, 16(1):21–34, 2006.
- Andrew Farmer. *HERMIT: Mechanized Reasoning during Compilation in the Glasgow Haskell Compiler*. PhD thesis, University of Kansas, 2015.
- Andrew Farmer, Christian Hoener zu Siederdisen, and Andy Gill. The HERMIT in the stream: Fusing stream fusion's concatmap. In *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation*, pages 97–108. ACM, 2014.

- Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. *SIGMOD Rec.*, 24(2):47–58, May 1995. ISSN 0163-5808. doi: 10.1145/568271.223789. URL <https://doi.org/10.1145/568271.223789>.
- Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, December 2000. ISSN 0362-5915. doi: 10.1145/377674.377676. URL <https://doi.org/10.1145/377674.377676>.
- Leonidas Fegaras and Md Hasanuzzaman Noor. Compile-time code generation for embedded data-intensive query languages. In *2018 IEEE International Congress on Big Data (BigData Congress)*, pages 1–8, 2018.
- Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI '93*, pages 237–247, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. doi: 10.1145/155090.155113. URL <http://doi.acm.org/10.1145/155090.155113>.
- Shaun A Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software (TOMS)*, 32(2):195–222, 2006.
- Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Formal loop merging for signal transforms. *PLDI '05*, pages 315–326. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065048. URL <http://doi.acm.org/10.1145/1065010.1065048>.
- Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. Operator language: A program generation framework for fast kernels. In *Domain-Specific Languages*, pages 385–409. Springer, 2009.
- Yoshihiko Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.
- Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *ACM SIGPLAN Notices*, volume 36, pages 74–85. ACM, 2001.
- Jacques Garrigue and Didier Rémy. Ambivalent types for principal type inference with GADTs. In *Asian Symposium on Programming Languages and Systems*, pages 257–272. Springer, 2013.
- Paolo G. Giarrusso. Open GADTs and Declaration-site Variance: A Problem Statement. In *Proceedings of the 4th Workshop on Scala, SCALA '13*, pages 5:1–5:4, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2064-1.
- Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. A language and program for complex Bayesian modelling. *The Statistician*, pages 169–177, 1994.

Bibliography

- Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. FPCA, pages 223–232. ACM, 1993.
- Andrew John Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, University of Glasgow, 1996.
- George Giorgidze, Torsten Grust, Nils Schweinsberg, and Jeroen Weijers. Bringing back monad comprehensions. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell ’11, page 13–22, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308601. doi: 10.1145/2034675.2034678. URL <https://doi.org/10.1145/2034675.2034678>.
- Michele Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*, pages 68–85. Springer, 1982.
- Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: A language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.
- Google. Protocol buffers. <https://github.com/protocolbuffers/protobuf>, 2008. Accessed: 2019-04-06.
- Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014.
- Samuel Gruetter. Connecting Scala to DOT. *MSc semester project, EPFL*, June 2016. URL <https://github.com/samuelgruetter/dot-calculus/blob/master/doc/Connecting-Scala-to-DOT/report.pdf>.
- Torsten Grust. *Monad Comprehensions: A Versatile Representation for Queries*, pages 288–311. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-662-05372-0. doi: 10.1007/978-3-662-05372-0_12. URL https://doi.org/10.1007/978-3-662-05372-0_12.
- Michael Haidl, Michel Steuwer, Tim Humernbrum, and Sergei Gorlatch. Multi-stage Programming for GPUs in C++ Using PACXX. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit, GPGPU ’16*, pages 32–41, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4195-0. doi: 10.1145/2884045.2884049. URL <http://doi.acm.org/10.1145/2884045.2884049>. event-place: Barcelona, Spain.
- G. W. Hamilton. Distillation: Extracting the essence of programs. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM ’07*, pages 61–70, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-620-2. doi: 10.1145/1244381.1244391. URL <http://doi.acm.org/10.1145/1244381.1244391>.
- Laurent Hascoet and Valérie Pascual. The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification. *ACM Trans. Math. Softw.*, 39(3):20:1–20:43, May 2013. ISSN 0098-3500.

- David Herman. *A theory of typed hygienic macros*. PhD thesis, Northeastern University, 2010.
- David Herman and Mitchell Wand. A theory of hygienic macros. In *European Symposium on Programming*, pages 48–62. Springer, 2008.
- Christian Hofer and Klaus Ostermann. Modular domain-specific language components in Scala. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE ’10, pages 83–92, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0154-1. doi: 10.1145/1868294.1868307. URL <http://doi.acm.org/10.1145/1868294.1868307>.
- 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 137–148. ACM, 2008.
- Robin J. Hogan. Fast Reverse-Mode Automatic Differentiation Using Expression Templates in C++. *ACM Trans. Math. Softw.*, 40(4):26:1–26:16, July 2014. ISSN 0098-3500.
- Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.
- Paul Hudak. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR ’98, pages 134–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8377-5.
- G rard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta informatica*, 11(1):31–55, 1978.
- Jun Inoue, Oleg Kiselyov, and Yuki Yoshi Kameyama. Staging Beyond Terms: Prospects and Challenges. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM ’16, pages 103–108, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4097-7. doi: 10.1145/2847538.2847548. URL <http://doi.acm.org/10.1145/2847538.2847548>.
- Jan Martin Jansen. Programming in the λ -calculus: From Church to Scott and back. In *Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday on The Beauty of Functional Code - Volume 8106*, pages 168–180, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-40354-5. doi: 10.1007/978-3-642-40355-2_12. URL https://doi.org/10.1007/978-3-642-40355-2_12.
- Dean F Jerding, John T Stasko, and Thomas Ball. Visualizing interactions in program executions. In *Proceedings of the 19th international conference on Software engineering*, pages 360–370. ACM, 1997.
- Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

Bibliography

- Simon L. Peyton Jones. Compiling haskell by program transformation: A report from the trenches. In HanneRiis Nielson, editor, *Programming Languages and Systems - ESOP '96*, volume 1058 of *Lecture Notes in Computer Science*, pages 18–44. Springer Berlin Heidelberg, 1996. ISBN 978-3-540-61055-7. doi: 10.1007/3-540-61055-3_27. URL http://dx.doi.org/10.1007/3-540-61055-3_27.
- Simon Peyton Jones and Philip Wadler. Comprehensive comprehensions. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, page 61–72, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936745. doi: 10.1145/1291201.1291209. URL <https://doi.org/10.1145/1291201.1291209>.
- Manohar Jonnalagedda and Sandro Stucki. Fold-based fusion as a library: A generative programming pearl. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, pages 41–50. ACM, 2015a. ISBN 978-1-4503-3626-0. doi: 10.1145/2774975.2774981. URL <http://doi.acm.org/10.1145/2774975.2774981>.
- Manohar Jonnalagedda and Sandro Stucki. Fold-based Fusion As a Library: A Generative Programming Pearl. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, SCALA 2015, pages 41–50, Portland, OR, USA, 2015b. ACM. ISBN 978-1-4503-3626-0.
- Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. Staged parser combinators for efficient data processing. In *Acm Sigplan Notices*, volume 49, pages 637–653. ACM, 2014.
- Nicolai M Josuttis. *The C++ standard library: a tutorial and reference*. Addison-Wesley, 2012.
- Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. Yin-Yang: Concealing the deep embedding of DSLs. GPCE 2014, pages 73–82. ACM, 2014.
- Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Combinators for impure yet hygienic code generation. *Science of Computer Programming*, 112 (part 2):120–144, November 2015. doi: 10.1016/j.scico.2015.08.007.
- Jerzy Karczmarczuk. Functional differentiation of computer programs. *ACM SIGPLAN Notices*, 34(1):195–203, 1999.
- Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. Just-in-time data virtualization: Lightweight data management with ViDa. In *CIDR*, 2015.
- Gershon Kedem. Automatic differentiation of computer programs. *ACM Trans. Math. Softw.*, 6 (2):150–165, June 1980. ISSN 0098-3500.
- Andrew Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 177–190, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291179. URL <http://doi.acm.org/10.1145/1291151.1291179>.

- Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 21–40, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0.
- Andrew Kennedy and Don Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 1–12, New York, NY, USA, 2001. ACM. ISBN 978-1-58113-414-8. doi: 10.1145/378795.378797. URL <http://doi.acm.org/10.1145/378795.378797>. event-place: Snowbird, Utah, USA.
- Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for Lisp-like multi-staged languages. In *ACM SIGPLAN Notices*, volume 41, pages 257–268. ACM, 2006.
- Oleg Kiselyov. Typed tagless final interpreters. In *Generic and Indexed Programming*, pages 130–174. Springer, 2012.
- Oleg Kiselyov. The design and implementation of BER MetaOCaml. In *International Symposium on Functional and Logic Programming*, pages 86–102. Springer, 2014.
- Oleg Kiselyov. MetaOCaml – an OCaml dialect for multi-stage programming, 2017. URL <https://web.archive.org/web/20170725111517/http://okmij.org/ftp/ML/MetaOCaml.html>.
- Oleg Kiselyov. Reconciling abstraction with high performance: A MetaOCaml approach. *Found. Trends Program. Lang.*, 5(1):1–101, June 2018. ISSN 2325-1107. doi: 10.1561/25000000038. URL <https://doi.org/10.1561/25000000038>.
- Oleg Kiselyov and Chung-Chieh Shan. Embedded probabilistic programming. In *Domain-Specific Languages*, pages 360–384. Springer, 2009.
- Oleg Kiselyov, Yuki Yoshi Kameyama, and Yuto Sudo. Refined environment classifiers. In *Asian Symposium on Programming Languages and Systems*, pages 271–291. Springer, 2016.
- Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 285–299. ACM, 2017.
- P. Klint, T. v. d. Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177, Sept 2009.
- Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.
- Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and*

Bibliography

- Implementation*, PLDI '94, pages 147–158, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X.
- Christoph Koch. Incremental query evaluation in a ring of databases. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 87–98. ACM, 2010.
- Christoph Koch. Abstraction without regret in database systems building: a manifesto. *IEEE Data Eng. Bull.*, 37(1):70–79, 2014.
- Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal*, 23(2):253–278, 2014. ISSN 1066-8888.
- Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP '86*, pages 151–161, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4.
- Pieter Koopman, Rinus Plasmeijer, and Jan Martin Jansen. Church Encoding of Data Types Considered Harmful for Implementations: Functional Pearl. In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages, IFL '14*, pages 4:1–4:12, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3284-2. doi: 10.1145/2746325.2746330. URL <http://doi.acm.org/10.1145/2746325.2746330>.
- Konstantinos Krikellias, Stratis Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *Proc. International Conference on Data Engineering (ICDE)*, pages 613–624, 2010.
- Tejas D Kulkarni, Pushmeet Kohli, Joshua B Tenenbaum, and Vikash Mansinghka. Picture: A probabilistic programming language for scene perception. In *Proceedings of the iee conference on computer vision and pattern recognition*, pages 4390–4399, 2015.
- HyounJoong Lee, Kevin J Brown, Arvind K Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.
- Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. Shallow embedding of DSLs via online partial evaluation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2015*, pages 11–20, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3687-1.
- Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless Gradients in Numpy. In *ICML 2015 AutoML Workshop*, 2015.
- Geoffrey Mainland. Why it's nice to be quoted: Quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop, Haskell '07*, pages 73–82, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-674-5.

- Geoffrey Mainland. Explicitly heterogeneous metaprogramming with MetaHaskell. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, page 311–322, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310543. doi: 10.1145/2364527.2364572. URL <https://doi.org/10.1145/2364527.2364572>.
- Vikash K Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 603–616. ACM, 2018.
- I. Masliah, M. Baboulin, and J. Falcou. Meta-programming and Multi-stage Programming for GPGPUs. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pages 369–376, September 2016. doi: 10.1109/MCSOC.2016.49.
- Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 482–494, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062380. URL <http://doi.acm.org/10.1145/3062341.3062380>.
- Conor McBride and James McKinna. Functional pearl: I am not a number—I am a free variable. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, page 1–9, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138504. doi: 10.1145/1017472.1017477. URL <https://doi.org/10.1145/1017472.1017477>.
- James McKinna and Robert Pollack. Pure type systems formalized. In *Typed Lambda Calculi and Applications*, pages 289–305. Springer, 1993.
- Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. SIGMOD '06, pages 706–706. ACM, 2006. ISBN 1-59593-434-0. doi: 10.1145/1142473.1142552. URL <http://doi.acm.org/10.1145/1142473.1142552>.
- Marjan Mernik. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments: Recent Developments*. IGI Global, 2012.
- Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L Ong, and Andrey Kolobov. BLOG: Probabilistic Models with Unknown Objects. *Statistical relational learning*, page 373, 2007.
- Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978. ISSN 0022-0000. doi: [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4). URL <http://www.sciencedirect.com/science/article/pii/0022000078900144>.
- Tom Minka, John Winn, John Guiver, and David Knowles. Infer.net 2.4, 2010. microsoft research cambridge, 2014.

Bibliography

- Mehryar Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002.
- Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, pages 117–120. ACM, 2012.
- Fabian Nagel, Gavin Bierman, and Stratis D. Viglas. Code generation for efficient query processing in managed runtimes. *Proc. VLDB Endow.*, 7(12):1095–1106, August 2014. ISSN 2150-8097.
- Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. Everything old is new again: Quoted domain-specific languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2016, pages 25–36, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4097-7.
- Aleksandar Nanevski. Meta-programming with names and necessity. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 206–217, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8.
- Aleksandar Nanevski and Frank Pfenning. Staged computation with names and necessity. *Journal of Functional Programming*, 15(6):893–939, 2005.
- Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)*, 9(3):23, 2008.
- Sri Hari Krishna Narayanan, Boyana Norris, and Beata Winnicka. ADIC2: Development of a component source transformation system for differentiating C and C++. *Procedia Computer Science*, 1(1):1845–1853, 2010.
- Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.
- Gregory Neverov and Paul Roe. Metaphor: A Multi-stage, Object-Oriented Programming Language. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering*, Lecture Notes in Computer Science, pages 168–185. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-30175-2.
- Abel Nieto. Towards Algorithmic Typing for DOT (Short Paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, SCALA 2017, pages 2–7, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5529-2. doi: 10.1145/3136000.3136003. URL <http://doi.acm.org/10.1145/3136000.3136003>.
- Martin Odersky and Adriaan Moors. Fighting bit Rot with Types (Experience Report: Scala Collections). In Ravi Kannan and K. Narayan Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 4 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 427–451, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-13-2.

- Martin Odersky and Matthias Zenger. Scalable Component Abstractions. In *OOPSLA*, pages 41–57, San Diego, CA, USA, 2005. ISBN 1-59593-031-0.
- Martin Odersky, Guillaume Martres, and Dmitry Petrashko. Implementing Higher-kinded Types in Dotty. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala, SCALA 2016*, pages 51–60, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4648-1. doi: 10.1145/2998392.2998400. URL <http://doi.acm.org/10.1145/2998392.2998400>.
- Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicity: Foundations and applications of implicit function types. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158130. URL <https://doi.org/10.1145/3158130>.
- Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in Scala: Towards the systematic construction of generators for performance libraries. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences, GPCE '13*, pages 125–134, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2373-4.
- Junpei Oishi and Yuki Yoshi Kameyama. Staging with Control: Type-safe Multi-stage Programming with Control Operators. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017*, pages 29–40, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5524-7. doi: 10.1145/3136040.3136049. URL <http://doi.acm.org/10.1145/3136040.3136049>. event-place: Vancouver, BC, Canada.
- Bruno C d S Oliveira and William R Cook. Extensibility for the masses. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2012.
- Bruno C.d.S. Oliveira, Meng Wang, and Jeremy Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, OOPSLA '08*, page 439–456, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582153. doi: 10.1145/1449764.1449799. URL <https://doi.org/10.1145/1449764.1449799>.
- Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10*, pages 341–360, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6.
- Nuno Oliveira, Maria João Pereira, Pedro Henriques, and Daniela Cruz. Domain specific languages: A theoretical survey. *INForum'09-Simpósio de Informática*, 2009.
- Lionel Parreaux and Christoph Koch. Comprehending monoids with class (extended abstract). 2018. URL <https://icfp18.sigplan.org/details/tyde-2018/12/Extended-Abstract-Comprehending-Monoids-with-Class>.

Bibliography

- Lionel Parreaux and Amir Shaikhha. Multi-stage programming in the large with staged classes. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2020, New York, NY, USA, 2020. ACM. doi: 10.1145/3425898.3426961. URL <https://doi.org/10.1145/3425898.3426961>.
- Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. Quoted Staged Rewriting: A practical approach to library-defined optimizations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2017, pages 131–145, New York, NY, USA, 2017a. ACM. ISBN 978-1-4503-5524-7. doi: 10.1145/3136040.3136043. URL <http://doi.acm.org/10.1145/3136040.3136043>.
- Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. Squid: Type-safe, hygienic, and reusable quasiquotes. In *Proceedings of the 2017 8th ACM SIGPLAN Symposium on Scala*, SCALA 2017. ACM, 2017b. ISBN 978-1-4503-5529-2/17/10.
- Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. Unifying analytic and statically-typed quasiquotes. *Proc. ACM Program. Lang.*, 2(POPL), December 2017c. doi: 10.1145/3158101. URL <https://doi.org/10.1145/3158101>.
- Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. Towards improved GADT reasoning in scala. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, Scala '19, page 12–16, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368247. doi: 10.1145/3337932.3338813. URL <https://doi.org/10.1145/3337932.3338813>.
- Barak A Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):7, 2008.
- Tomas Petricek, Gustavo Guerra, and Don Syme. Types from Data: Making Structured Data First-class Citizens in F#. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 477–490, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908115. URL <http://doi.acm.org/10.1145/2908080.2908115>. event-place: Santa Barbara, CA, USA.
- Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. *ACM SIGPLAN*, September 2001.
- Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical Report 137, Charles River Analytics, 2009.
- Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Notices*, volume 23, pages 199–208. ACM, 1988.
- Randy Pollack, Masahiko Sato, and Wilmer Ricciotti. A canonical locally named representation of binding. *Journal of Automated Reasoning*, 49(2):185–207, 2012.

- F. Pottier. Static name control for FreshML. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 356–365, 2007.
- François Pottier. An overview of caml. *Electronic Notes in Theoretical Computer Science*, 148 (2):27 – 52, 2006. ISSN 1571-0661. Proceedings of the ACM-SIGPLAN Workshop on ML (ML 2005).
- Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.
- WV Quine. Mathematical logic. 1940.
- R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014. URL <http://www.R-project.org/>.
- Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, page 519–530, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320146. doi: 10.1145/2491956.2462176. URL <https://doi.org/10.1145/2491956.2462176>.
- Marianna Rapoport and Ondřej Lhoták. A path to DOT: Formalizing fully-path-dependent types. 2019. URL <http://arxiv.org/abs/1904.07298>.
- Nico Reißmann. Utilizing the value state dependence graph for haskell. 2012.
- Morten Rhiger. First-class open and closed code framgments. *Trends in Functional Programming*, 6:127–144, 2005.
- Morten Rhiger. *Programming Languages and Systems: 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, chapter Staged Computation with Staged Lexical Scope, pages 559–578. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012a. ISBN 978-3-642-28869-2.
- Morten Rhiger. Hygienic quasiquotation in Scheme. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, pages 58–64. ACM, 2012b.
- Tiark Rompf. Reflections on LMS: exploring front-end alternatives. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, pages 41–50. ACM, 2016.
- Tiark Rompf and Nada Amin. From F to DOT: Type Soundness Proofs with Definitional Interpreters. October 2015a. URL <https://arxiv.org/abs/1510.05216v2>.

Bibliography

- Tiark Rompf and Nada Amin. Functional pearl: a SQL to C compiler in 500 lines of code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 2–9, 2015b.
- 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, GPCE '10*, pages 127–136, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0154-1. doi: 10.1145/1868294.1868314. URL <http://doi.acm.org/10.1145/1868294.1868314>.
- Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Building-blocks for performance oriented DSLs. *Electronic Proceedings in Theoretical Computer Science*, 66:93–117, Sep 2011. ISSN 2075-2180. doi: 10.4204/eptcs.66.5. URL <http://dx.doi.org/10.4204/EPTCS.66.5>.
- Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *POPL*, pages 497–510, 2013.
- Tiark Rompf, Nada Amin, Thierry Coppey, Mohammad Dashti, Manohar Jonnalagedda, Yannis Klonatos, Martin Odersky, and Christoph Koch. Abstraction without regret for efficient data processing. In *Data-Centric Programming Workshop*, 2014.
- B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '88*, pages 12–27, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73562. URL <http://doi.acm.org/10.1145/73560.73562>.
- Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *ECOOP*, pages 258–282. Springer, 2012.
- Adrian Sampson, Kathryn S. McKinley, and Todd Mytkowicz. Static Stages for Heterogeneous Programming. *Proc. ACM Program. Lang.*, 1(OOPSLA):71:1–71:27, October 2017. ISSN 2475-1421. doi: 10.1145/3133895. URL <http://doi.acm.org/10.1145/3133895>.
- Taisuke Sato. A glimpse of symbolic-statistical modeling by PRISM. *Journal of Intelligent Information Systems*, 31(2):161–176, October 2008.
- Yuhi Sato, Yuki Yoshi Kameyama, and Takahisa Watanabe. Module generation without regret. In *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2020*, page 1–13, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370967. doi: 10.1145/3372884.3373160. URL <https://doi.org/10.1145/3372884.3373160>.

- Gabriel Scherer and Didier Rémy. GADTs Meet Subtyping. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 554–573. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-37036-6.
- Maximilian Scherr and Shigeru Chiba. Implicit staging of EDSL expressions: A bridge between shallow and deep embedding. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28 – August 1, 2014. Proceedings*, pages 385–410, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44202-9.
- Maximilian Scherr and Shigeru Chiba. Almost first-class language embedding: Taming staged embedded DSLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 21–30, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3687-1.
- Denys Shabalin. Hygiene for scala. Technical report, 2014. URL <http://infoscience.epfl.ch/record/215109>.
- Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for Scala. Technical report, 2013. URL <http://infoscience.epfl.ch/record/185242>.
- Amir Shaikhha and Lionel Parreaux. Finally, a Polymorphic Linear Algebra Language (Pearl). In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:29, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-111-5. doi: 10.4230/LIPIcs.ECOOP.2019.25. URL <http://drops.dagstuhl.de/opus/volltexte/2019/10817>.
- Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to architect a query compiler. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD ’16, pages 1907–1922, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7.
- Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, pages 12–23, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5181-2.
- Amir Shaikhha, Mohammad Dashti, and Christoph Koch. Push versus Pull-Based Loop Fusion in Query Engines. *Journal of Functional Programming*, 28:e10, 2018a.
- Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, Simon Peyton Jones, and Christoph Koch. Efficient differentiable programming in a functional array-processing language. *arXiv preprint arXiv:1806.02136*, 2018b.
- Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell ’02, pages 1–16. ACM, 2002.

Bibliography

- Tim Sheard, Zine-el-abidine Benaissa, and Emir Pasalic. DSL implementation using staging and monads. In *Proceedings of the 2nd Conference on Domain-specific Languages*, DSL '99, pages 81–94, New York, NY, USA, 1999. ACM. ISBN 1-58113-255-7.
- Tim Sheard, James Hook, and Nathan Linger. GADTs + extensible kinds = dependent programming, 2005.
- Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: Programming with binders made simple. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 263–274, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7. doi: 10.1145/944705.944729.
- AnthonyM. Sloane. Lightweight language processing in kama. In JoãoM. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 408–425. 2011.
- Morten H Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In *Proceedings of ILPS'95, the International Logic Programming Symposium*. Citeseer, 1995.
- Daniele G. Spampinato and Markus Püschel. A basic linear algebra compiler. CGO '14, pages 23:23–23:32. ACM, 2014. ISBN 978-1-4503-2670-4. doi: 10.1145/2544137.2544155. URL <http://doi.acm.org/10.1145/2544137.2544155>.
- James Stanier. Removing and restoring control flow with the value state dependence graph. Master's thesis, University of Sussex, 2012.
- James Stanier and Des Watson. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv.*, 45(3):26:1–26:27, July 2013. ISSN 0360-0300.
- Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code. *ACM SIGPLAN Notices*, 50(9):205–217, 2015.
- Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005. ISBN 1-59593-154-6.
- Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. A practical unification of multi-stage programming and macros. *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, page 14, 2018. doi: 10.1145/3278122.3278139. URL <http://infoscience.epfl.ch/record/257176>.

- Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.
- Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: Generating a high performance DSL implementation from a declarative specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, page 145–154, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323734. doi: 10.1145/2517208.2517220. URL <https://doi.org/10.1145/2517208.2517220>.
- Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. *System F with type equality coercions*. January 2011. ISBN 978-1-59593-393-5. URL <https://www.microsoft.com/en-us/research/publication/system-f-with-type-equality-coercions/>.
- Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. ICFP '02, pages 124–132. ACM, 2002. ISBN 1-58113-487-8. doi: 10.1145/581478.581491. URL <http://doi.acm.org/10.1145/581478.581491>.
- Bo Joel Svensson and Josef Svenningsson. Defunctionalizing push arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, pages 43–52, NY, USA, 2014. ACM. ISBN 978-1-4503-3040-4.
- Kedar Swadi, Walid Taha, Oleg Kiselyov, and Emir Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '06, page 160–169, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595931961. doi: 10.1145/1111542.1111570. URL <https://doi.org/10.1145/1111542.1111570>.
- Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, Wonseok Chae, Uladzimir Matsveyeu, and Tomas Petricek. Strongly-typed language support for internet- scale information sources.
- Donald Syme. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 Workshop on ML*. ACM, 2006.
- Walid Taha. *Multi-stage programming: Its theory and applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.
- Walid Taha. *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*, chapter A Gentle Introduction to Multi-stage Programming, pages 30–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-25935-0.

Bibliography

- Walid Taha and Michael Florentin Nielsen. Environment classifiers. *SIGPLAN Not.*, 38(1): 26–37, January 2003. ISSN 0362-1340.
- Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Notices*, volume 32, pages 203–217. ACM, 1997.
- Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.
- Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18*, pages 307–322, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4703-7. doi: 10.1145/3183713.3196893. URL <http://doi.acm.org/10.1145/3183713.3196893>. event-place: Houston, TX, USA.
- Transaction Processing Performance Council. *TPC-H, an Ad-Hoc, Decision Support Benchmark*. 1999. URL <http://www.tpc.org/tpch>.
- Phil Trinder. Comprehensions, a query notation for DBPLs. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types & Persistent Data: Bulk Types & Persistent Data*, DBPL3, page 55–68, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN 1558602429.
- Valentin F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, pages 481–509, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-70589-5.
- Vlad Ureche, Cristian Talau, and Martin Odersky. Miniboxing: Improving the speed to code size tradeoff in parametric polymorphism translations. In *Proceedings of the ACM SIGPLAN 2013 Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'13)*, number CONE, 2013.
- Vlad Ureche, Aggelos Biboudis, Yannis Smaragdakis, and Martin Odersky. Automating ad hoc data representation transformations. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015*, pages 801–820, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5.
- David Vandevoorde and Nicolai M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 978-0-201-73484-3.
- Todd L. Veldhuizen. C++ Templates as Partial Evaluation. *arXiv:cs/9810010*, October 1998. URL <http://arxiv.org/abs/cs/9810010>. arXiv: cs/9810010.
- Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries, 1998.
- Stratis Viglas, Gavin M. Bierman, and Fabian Nagel. Processing declarative queries through generating imperative code in managed runtimes. *IEEE Data Eng. Bull.*, 37(1):12–21, 2014.

- Eelco Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57:109–143, 2001.
- Eelco Visser. Meta-programming with concrete object syntax. In *Proc. International Conference on Generative Programming and Component Engineering (GPCE)*, pages 299–315. Springer, 2002.
- Eelco Visser, Zine-el-Abidine Benaïssa, and Andrew Tolmach. Building program optimizers with rewriting strategies. ICFP '98, pages 13–26, 1998. ISBN 1-58113-024-4. doi: 10.1145/289423.289425. URL <http://doi.acm.org/10.1145/289423.289425>.
- Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP'88*, pages 344–358. Springer, 1988.
- Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, page 61–78, New York, NY, USA, 1990. Association for Computing Machinery. ISBN 089791368X. doi: 10.1145/91556.91592. URL <https://doi.org/10.1145/91556.91592>.
- Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.
- Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, November 2015. ISSN 0001-0782. doi: 10.1145/2699407. URL <https://doi.org/10.1145/2699407>.
- Fei Wang, James Decker, Xilun Wu, Gregory Essertel, and Tiark Rompf. Backpropagation with callbacks: Foundations for efficient and expressive differentiable programming. In *Advances in Neural Information Processing Systems*, pages 10200–10211, 2018.
- Takahisa Watanabe and Yuki Yoshi Kameyama. Program generation for ML modules (short paper). In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '18, page 60–66, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450355872. doi: 10.1145/3162072. URL <https://doi.org/10.1145/3162072>.
- Radosław Waśko. Formal foundations for GADTs in scala. 2020. URL <http://infoscience.epfl.ch/record/277075>.
- Matthew J Weinstein and Anil V Rao. Algorithm 984: ADiGator, a toolbox for the algorithmic differentiation of mathematical functions in MATLAB using source transformation via operator overloading. *ACM Trans. Math. Softw.*, 2016.
- Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. *ACM Sigplan Notices*, 45(6): 400–411, 2010.
- Thomas Würthinger. Extending the graal compiler to optimize libraries. In *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 41–42. ACM, 2011.

Bibliography

- Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 224–235, New York, NY, USA, 2003. ACM. ISBN 1-58113-628-5. doi: 10.1145/604131.604150. URL <http://doi.acm.org/10.1145/604131.604150>.
- Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. SPL: A Language and Compiler for DSP Algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 298–308, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2.
- Jeremy Yallop. Staged Generic Programming. *Proc. ACM Program. Lang.*, 1(ICFP):29:1–29:29, August 2017. ISSN 2475-1421.
- Jeremy Yallop and Oleg Kiselyov. Generating mutually recursive definitions. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2019, page 75–81, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362269. doi: 10.1145/3294032.3294078. URL <https://doi.org/10.1145/3294032.3294078>.
- Jeremy Yallop and Leo White. Modular macros. In *OCaml Users and Developers Workshop*, volume 6, 2015.
- Yizhou Zhang and Andrew C. Myers. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi: 10.1145/3290318. URL <https://doi.org/10.1145/3290318>.

Lionel Parreaux | Curriculum vitae

Route de Chavannes 68 – 1007 Lausanne – Switzerland

☎ +41 79 135 48 94 • ✉ lionel.parreaux@gmail.com
🌐 lptk.github.io/about • in [lparreaux](#) • 🌐 LPTK

Education

EPFL (Swiss Federal Institute of Technology) <i>Ph.D. in Computer Science</i>	Lausanne 2014–2020
NUS (National University of Singapore) <i>Academic exchange (1 semester)</i>	Singapore Fall 2013
INSA Lyon (National Institute of Applied Science) <i>M.Sc. (Engineering Degree) in Computer Science</i>	Lyon 2009–2014

Languages: French (mother tongue) English (bilingual) Spanish (intermediate)

Research interests: Programming languages, type systems, compiler design, domain-specific languages, and database technology. I believe that improving the performance, safety, and usability of high-level programming is essential to the future of software engineering as a whole.

Publications

- **Lionel Parreaux** and Amir Shaikhha. 2020. **Multi-stage Programming in the Large with Staged Classes**. In Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (**GPCE 2020**). DOI: <https://doi.org/10.1145/3425898.3426961>
- **Lionel Parreaux**. 2020. **The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl)**. In Proc. ACM Program. Lang. 4, ICFP, Article 124 (**ICFP 2020**). DOI: <https://doi.org/10.1145/3409006>
- Amir Shaikhha and **Lionel Parreaux**. 2019. **Finally, a Polymorphic Linear Algebra Language**. In 33rd European Conference on Object-Oriented Programming (**ECOOP 2019**). DOI: <https://doi.org/10.4230/LIPIcs.ECOOP.2019.25>
- **Lionel Parreaux**, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. 2019. **Towards improved GADT reasoning in Scala**. In Proceedings of the Tenth ACM SIGPLAN Symposium on Scala (**SCALA 2019**). DOI: <https://doi.org/10.1145/3337932.3338813>
- **Lionel Parreaux** and Christoph E. Koch. 2018. **Comprehending Monoids with Class** (Extended Abstract). In Proceedings of Type-Driven Development (**TyDe 2018**). <https://icfp18.sigplan.org/details/tyde-2018/12>
- **Lionel Parreaux**, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2018. **Unifying Analytic and Statically-Typed Quasiquotes**. In Proc. ACM Program. Lang. (**POPL 2018**). DOI: <https://doi.org/10.1145/3158101>
- **Lionel Parreaux**, Amir Shaikhha, and Christoph E. Koch. 2017. **Quoted staged rewriting: a practical approach to library-defined optimizations**. In Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (**GPCE 2017**). DOI: <https://doi.org/10.1145/3136040.3136043>
- **Lionel Parreaux**, Amir Shaikhha, and Christoph E. Koch. 2017. **Squid: type-safe, hygienic, and reusable quasiquotes**. In Proceedings of the 8th ACM SIGPLAN International Symposium on Scala (**SCALA 2017**). DOI: <https://doi.org/10.1145/3136000.3136005> 323

- Amir Shaikhha, Yannis Klonatos, **Lionel Parreaux**, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. **How to Architect a Query Compiler**. In Proceedings of the 2016 International Conference on Management of Data (**SIGMOD 2016**). DOI: <https://doi.org/10.1145/2882903.2915244>

Recognition

Awards.....

- (2017) **GPCE Best Paper Award** (*Quoted Staged Rewriting* paper).
- (2014) EPFL **EDIC PhD program fellowship**.

Presentations, Seminars, and Invitations.....

- (July 2019) *Towards improved GADT reasoning in Scala*. Conference talk, SCALA.
- (June 2018) *Fearless Metaprogramming with Squid*. Invited talk, DIMA lab, TU Berlin.
- (June 2018) *Fearless Metaprogramming with Squid*. Invited talk, Amazon Berlin.
- (September 2018) *Comprehending Monoids with Class*. Type-Driven Development, St. Louis.
- (January 2018) *Unifying analytic and statically-typed quasiquotes*. Conference talk, POPL.
- (December 2017) *Unifying analytic and statically-typed quasiquotes*. Invited talk, EPFL LAMP.
- (October 2017) *Quoted Staged Rewriting: a Practical Approach to Library-Defined Optimizations*. Conference talk, GPCE.
- (October 2017) *Squid: Type-Safe, Hygienic, and Reusable Quasiquotes*. Conference talk, SCALA.
- (September 2017) *Quoted Staged Rewriting: a Practical Approach to Library-Defined Optimizations*. Invited talk, EPFL LAMP.
- (2017, 2018, 2019) Google Compiler and Programming Language Summit, Munich.
- (2016) Google PhD Student Summit on Compiler & Programming Technology, Munich.

Open Source Contributions.....

- **Simple-sub** (52 stars)  <https://github.com/LPTK/simple-sub>
- **Squid** (171 stars)  <https://github.com/epfldata/squid>
- **dbStage** (12 stars)  <https://github.com/epfldata/dbstage>
- **Boilerless** (38 stars)  <https://github.com/lptk/boilerless>

References.....

- **Christoph E. Koch**, EPFL, Lausanne. christoph.koch@epfl.ch
<https://people.epfl.ch/christoph.koch>
- **Simon Peyton Jones**, Microsoft Research, Cambridge. simonpj@microsoft.com
<https://www.microsoft.com/en-us/research/people/simonpj/>
- **Martin Odersky**, EPFL, Lausanne. martin.odersky@epfl.ch
<https://lampwww.epfl.ch/~odersky/>
- **Viktor Kuncak**, EPFL, Lausanne. viktor.kuncak@epfl.ch
<http://lara.epfl.ch/~kuncak/>

Experience

Research.....

Research Intern, Optimization and Spreadsheets

EPFL, Lausanne

Microsoft Research, Cambridge

Summer 2018 (3 months)

- Started the design and implementation of a novel intermediate representation (IR) for optimizing pure functional languages, based on a graph representation with incremental substitution constructs.
- Designed a domain-specific language (DSL) for dynamic programming, as a way to capture common patterns of spreadsheet formulae, and worked on using the graph IR to optimize that DSL.

PhD Semester Project, Metaprogramming Tools

EPFL, Lausanne

Data Analysis Theory and Applications Laboratory (DATA Lab)

Spring 2015

- Implemented a quasiquotation engine for SC (Systems/Compiler co-design framework written in *Scala*), making use of advanced macros and type introspection.
- Benchmarked the macro implementation to optimize hot spots and enhance the user experience.

Report [available on this link](#).

PhD Semester Project, Type Systems

EPFL, Lausanne

Lab for Automated Reasoning and Analysis (LARA Lab)

Fall 2014

- Formalized a novel Type and Effect System based on regular-expression regions.
- Proved its safety regarding memory management (no dangling pointers; no memory leaks).

Report [available on this link](#).

Research Intern, Type Systems

EPFL, Lausanne

Lab for Automated Reasoning and Analysis (LARA)

Summer 2014 (5 months)

- Explored the design and implementation of *Seagl*, a programming language I designed allowing safe memory management without garbage collection, thanks to an effect system. The report is [available on this link](#).

Industry.....

R&D Intern, C++ Research Engineering

Palaiseau

Thales Research & Technology

Summer 2013 (3 months)

- Refactored and improved the usability of the open source *C++* library *paradiseo*.¹
- Implemented/tested new simulated annealing algorithm from a research paper; it was added to *paradiseo*.

Developer Intern, Build Systems and Testing

Boulogne Billancourt

DxO Labs (Image processing software)

Summer 2012 (3 months)

- Set up a regression testing framework in *Python*, generating results in *HTML5*.
- Coded a build system using *CMake* and *Python*.
- Integrated these critical tools, significantly increasing the productivity of core developers (8 to 10 persons).

Teaching Assistantship.....

CS-452: Foundations of Software (2019)

CS-449: Systems for Data Science (2018, 2019)

CS-210: Functional Programming (2018)


CS-251: Theory of Computation (2017)

CS-422: Database Systems (2016)

CS-110: Information, Computation, Communication (2016)

CS-111: Programming I (2015, 2017)

MATH-186: Mathematics II (2015)

¹ Paradiseo:  <https://github.com/nojhan/paradiseo>

Notable projects.....

- (2017–Present) Implementing **dbStage**,² a **staged database compilation framework** based on Squid. The goal of dbStage is to allow programmers to embed low-footprint database systems right inside their applications, with no impedance mismatch, all the while benefitting from the usual advanced database optimization techniques.
- (2016–Present) Developed the **Squid**³ **type-safe metaprogramming** framework for Scala, which extends the state of the art in multi-stage programming in several directions: it allows for **pattern matching and rewriting** existing code; it guarantees **type- and scope-safety** of metaprograms; it adds support for manipulating not only expressions but also definitions like classes and methods.
- (2018) As part of the **Microsoft Hackathon 2018**, created a language called MLScript which implemented **MLsub type inference**, compiled to Javascript, and could extract type information from TypeScript libraries for interoperability.
- (2016) Developed **Boilerless**,⁴ a macro annotation that makes defining Scala class hierarchies more concise, and which influenced the design of the later enum syntax in Scala 3.
- (2014) **Led development** of the final subject of the *Cod'INSA 2014 programming contest* – a real-time multiplayer game interacting with *Java*, *Python* and *C++* artificial intelligences written by the candidates, using *Apache Thrift*, *Swing*, and a web-based interface.⁵
The project was a success and allowed ranking the different teams according to their results.

References.....

Available on page 2.

² dbStage: <https://github.com/epfldata/dbstage>

³ Squid: <https://github.com/epfldata/squid>

⁴ Boilerless: <https://github.com/LPTK/Boilerless>

⁵ Cod'INSA final 2014: <https://github.com/cod-insa/cod-insa-2014>