KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Extensions to Type Classes and Pattern Match Checking

Georgios Karachalias

Supervisor:
Prof. dr. ir. Tom Schrijvers

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

May 2018

Extensions to Type Classes and Pattern Match Checking

Georgios KARACHALIAS

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

May 2018

Preface

Acknowledgments. First of all, I would like to thank my supervisor, Tom Schrijvers. Your excellent academic advising and the time you have invested in developing my skills made me the researcher I am today. I cannot thank you enough.

Next, I would like to thank all my jury members, as well as Imke van Steenkiste, Gert-Jan Bottu, and Ruben Pieters for revising this text.

I would also like to express my profound gratitude to the seasoned researchers I have worked with: Dimitrios Vytiniotis, Simon Peyton Jones, Bruno Oliveira, Philip Wadler, and Matija Pretnar. I am forever in debt to you for all the insight you so graciously gave me.

I cannot omit from this list the younger researchers I have worked with during the last four years: Gert-Jan Bottu, Koen Pauwels, and Michiel Derhaeg. For all your ideas, enthousiasm, and eagerness to follow me on the beautiful but difficult path of research. My only hope is that I have managed to impart to you at least a fragment of the knowledge that has been passed on to me; we should always pass on what we have learned.

I am also grateful to Ben Gamari for his generous support during my first steps in the GHC development process.

Many thanks go to my colleagues Amr Hany Saleh, Klara Marntirosian, Alexander Vandenbroucke, Amin Timany, Jesper Cockx, and Dominique Devriese, for the beautiful working environment we managed to create in the last four years, as well as the experiences we had outside the office. I would really like to preserve this.

Karin Michiels, you have my gratitude for making everything seem possible, without ever asking for anything in return. Our department at KU Leuven would not be the same without you.

I would like to specially thank Dimitrios Vytiniotis once more; though brief, you mentorship has played a major role in shaping my perspective, in life and academia in particular. I hope I can someday offer the same to someone else.

My friends Virginia Karachalia, George Chatzopoulos, Leonidas Labropoulos, Eleni Dourou, Erika Kotrotsou, Chryssa Zerva, Vicky Kontoura, Nikela Papadopoulou, George Psaropoulos, Theodwra Vergou, and Ioanna Alifieraki; thank you for being there for me.

I would also like to thank Katina, Lakis, and Panagiotis D. Vogazianos. Your long-term support helped me get through some of the most difficult phases of my life and aim for a Ph.D. in the first place. Katina and Panagiotis, I wish you were both alive to see this thesis come to completion.

I owe some special thanks to the following four:
Steven, for showing me how to live, Eleftherios, for showing me how to think, Foteini, for showing me how to care, and Erik, for showing me how to be myself.

Finally, I would like to dedicate this thesis to my family: Imke and Dharma.

Georgios Karachalias
May 2018

Abstract

Static typing is one of the most prominent techniques in the design of programming languages for making software more safe and more reusable. Furthermore, it provides opportunities for compilers to automatically generate boilerplate code, using mathematical foundations. In this thesis, we extend upon the design of Haskell, a general-purpose functional programming language with strong static typing, to offer more opportunities for reasoning, abstraction, and static code generation. More specifically, we improve upon two features: *pattern matching* and *type classes*.

Pattern matching denotes the act of performing case-analysis on the shape of a value, thus enabling a program to behave differently, depending on input information. With the advent of extensions such as Generalized Algebraic Data Types (GADTs), pattern guards, and view patterns, the task of reasoning about pattern matching has become much more complex. Though existing approaches can deal with some of these features, no existing algorithm can accurately reason about pattern matching in the presence of all of them, thus hindering the ability of the compiler to guide the development process. The first part of this thesis presents a short, easy-to-understand, and modular algorithm which can reason about lazy pattern matching with GADTs and guards. We have implemented our algorithm in the Glasgow Haskell Compiler.

The second part of this thesis extends upon the design of type classes, one of the most distinctive features of Haskell for function overloading and type-level programming. We develop three independent extensions that lift the expressive power of type classes from simple Horn clauses to a significant fragment of first-order logic, thus offering more possibilities for expressive type-level programming and automated code generation.

The first feature, *Quantified Class Constraints*, lifts the language of constraints from simple Horn clauses to Harrop formulae. It significantly increases the modelling power of type classes, while at the same time it enables a terminating type class resolution for a larger class of applications.

The second feature, *Functional Dependencies*, extends type classes with implicit type-level functions. Though functional dependencies have been implemented in Haskell compilers for almost two decades, several aspects of their semantics have been ill-understood. Thus, existing implementations of functional dependencies significantly deviate from their specification. We present a novel formalization of functional dependencies that addresses all such problems, and give the first type inference algorithm for functional dependencies that successfully elaborates the feature into a statically-typed intermediate language.

The third feature, *Bidirectional Instances*, allows for the interpretation of class instances bidirectionally, thus indirectly adding the biconditional connective in the language of constraints. This extension significantly improves the interaction of GADTs and type classes, since it allows functions with qualified types to perform structural induction over indexed types. Moreover, under this interpretation class-based extensions such as functional dependencies and associated types become much more expressive.

In summary, this thesis extends upon existing and develops new type-level features, promoting the usage of rich types that can capture and statically enforce program properties.

Beknopte Samenvatting

Static typing is een van de meest gebruikte technieken in het ontwerp van programmeertalen om software veiliger en meer herbruikbaar te maken. Verder geeft het *compilers* de mogelijkheid om automatisch *boilerplate* code te genereren, met behulp van wiskundige grondbeginselen. In deze thesis breiden we het ontwerp uit van Haskell, een functionele programmeertaal voor algemene doeleinden met sterke static typing, om meer mogelijkheiden tot (logische) redenering, abstractie, en statische generatie van code te kunnen bieden. Meer specifiek verbeteren we twee taalconcepten: *pattern matching* en *type classes.*

Pattern matching is het uitvoeren van een gevalsanalyse op de vorm van een waarde. Hierdoor kan een programma verschillende gedragen vertonen, afhankelijk van de ingevoerde informatie. Door de opkomst van extensies zoals *Generalised Algebraic Data Types* (GADTs), *pattern guards*, en *view patterns* is het redeneren over pattern matching veel complexer geworden. Hoewel bestaande aanpakken al om kunnen gaan met sommige van deze taalconcepten, is er geen bestaand algoritme dat precieze redeneringen kan geven over pattern matching met inachtneming van alle eigenschappen, waardoor het vermogen van de compiler om het ontwikkelingsproces te sturen gehinderd wordt. Het eerste deel van deze thesis stelt een kort, begrijpelijk, en modulair algoritme voor, dat kan redeneren over *lazy pattern matching* met GADTs en *guards.* We hebben ons algoritme geïmplementeerd in de Glasgow Haskell Compiler.

Het tweede deel van deze thesis geeft een uitbreiding van het ontwerp van type classes, een van de meest karakteristieke taalconcepten van Haskell voor function overloading en het programmeren op type-niveau. We ontwikkelen drie onafhankelijke extensies die de expressieve kracht van type classes verheffen van het niveau van simpele Horn-clausules naar een significant gedeelte van eerste-orde-logica, waardoor we meer mogelijkheden voor expressieve programmering op type-niveau, en geautomatiseerde generatie van code bieden.

Het eerste taalconcept, *Quantified Class Constraints*, tilt de taal van constraints van simpele Horn-clausules naar Harrop formules. Dit zorgt voor een significante toename in de modelleringskracht van type classes, en tegelijkertijd maakt het terminatie voor type class resolutie mogelijk voor een grotere klasse van applicaties.

Het tweede taalconcept, *Functional Dependencies*, breidt type classes uit met impliciete functies op type-niveau. Hoewel functional dependencies al bijna twee decennia geïmplementeerd zijn in Haskell compilers, vatten bestaande implementaties niet alle aspecten van de semantiek. Zodoende wijken de bestaande implementaties van functional dependencies significant af van hun specificaties. Wij presenteren een nieuwe formalisatie van functional dependencies die al deze problemen aanpakt, en we geven het eerste type inference algoritme voor functional dependencies dat deze taalconcept succesvol uitwerkt tot een *statically-typed* intermediaire taal.

Het derde taalconcept, *Bidirectional Instances*, staat een interpretatie van class instances in beide richtingen toe, waardoor het indirect een biconditionaire verbinding toevoegt aan de taal van constraints. Deze extensie verbetert de interactie van GADTs en type classes significant, omdat het functies met *qualified types* toestaat om structurele inductie uit te voeren over *indexed types*. Bovendien worden class-based extensies zoals functional dependencies en *associated types* door deze interpretatie veel expressiever.

Samengevat, deze thesis presenteert uitbreidingen van bestaande, en ontwikkeling van nieuwe type-level taalconcepten. Deze taalconcepten bevorderen het gebruik van *rich types* die programma-eigenschappen kunnen vatten, en statisch kunnen toepassen.

Translated from the English abstract by Imke van Steenkiste and revised by Tom Schrijvers.

Contents

Abstract iii

Contents vii

1 Introduction 1

1.1 Type Systems 1

1.2 Thesis Overview and Scientific Output 6

I Pattern Match Checking 11

2 Background 12

2.1 Algebraic Data Types and Pattern Matching 12

2.2 Pattern Matching 14

2.3 Generalized Algebraic Data Types 19

2.4 Problem Statement 20

3 Pattern Match Checking 29

3.1 Our Approach 29

3.2 Phase 1: Pattern Desugaring 33

3.3 Phase 2: Pattern Match Checking 38

3.4 Meta-theory 50

3.5 Related Work 54

3.6 Scientific Output 60

4 GHC Implementation 63

4.1 Alternative Formalization 64

4.2 Empty Case Expressions 69

4.3 Performance Improvements 75

4.4 The Oracle 80

4.5 Evaluation 82

4.6 Scientific Output 85

II Type Classes 87

5 Background 88

5.1 Polymorphism 88

5.2 System F: The Polymorphic Lambda Calculus 91

5.3 Type Reconstruction and Elaboration 95

5.4 Type Classes: Ad-hoc Polymorphism in Haskell 107

6 The Basic System 117

6.1 Extended System F 117

6.2 Syntax 121

6.3 Typing and Elaboration into System F 123

6.4 Type Inference with Elaboration into System F 127

6.5 Meta-theoretical Properties 135

6.6 Scientific Output 137

7 Quantified Constraints 139

7.1 Motivation . . . 139
7.2 Declarative Specification . . . 144
7.3 Type Inference with Elaboration . . . 152
7.4 Termination of Resolution . . . 157
7.5 Related Work . . . 159
7.6 Scientific Output . . . 162

8 Functional Dependencies 165

8.1 Motivation . . . 166
8.2 Logical Reading of Functional Dependencies . . . 172
8.3 Type Checking . . . 177
8.4 Target Language: System $\mathrm{F_C}$ . . . 182
8.5 Type Inference and Elaboration into System $\mathrm{F_C}$ . . . 189
8.6 Meta-theory . . . 198
8.7 Related Work . . . 204
8.8 Scientific Output . . . 205

9 Bidirectional Instances 207

9.1 Motivation . . . 207
9.2 Technical Challenges . . . 212
9.3 Bidirectional Instances . . . 216
9.4 Meta-theory . . . 220
9.5 Scientific Output . . . 222

10 Conclusion 225

10.1 Aim of the Thesis . . . 225
10.2 Summary . . . 226

10.3 Ongoing and Future Work 230

A Basic System: Additional Judgments **239**

A.1 Program and Declaration Typing 239

B Functional Dependencies: Additional Material **241**

B.1 Constraint Schemes, CHRs and System F_C 241

B.2 Poly-kinded, Generic Type Projections 244

B.3 Elaboration of Top-level Value Bindings 245

Index **247**

Bibliography **251**

List of Publications **261**

Chapter 1

Introduction

"In the beginning, the Universe was created. This had made a lot of people very angry and been widely regarded as a bad move."

—Douglas Adams, The Hitchhiker's Guide to the Galaxy, Vol. II

Our society's need for software has grown exponentially over the past years: from planes to mobile phones, most everyday tasks nowadays rely heavily on programs. As a consequence, both the size and the complexity of all these systems follows the same exponential curve. Unfortunately, the more complex the software, the more difficult it is to develop, maintain, reason about, and extend. Hence, the development process has become more slow and error-prone, pointing at a dire need for means to improve programmer productivity. One of the most common techniques in the design of programming languages for achieving reusability (and thus shorter and more concise code) while at the same time eliminating software bugs early in the development process is through *type systems*.

1.1 Type Systems

The term "type system" refers to the formal classification of language objects in different categories, each category supporting a limited set of operations. Such a category is referred to as a *type*: a type can be understood as the set of all expressions it classifies. For example, a numeric literal 5 can be assigned type *Int* (integer), while a character literal '*c*' can be assigned type *Char* (character).

More importantly, a type system also specifies a set of rules (*typing rules*) that assign types to more complex structures (e.g., functions, modules, etc.), using the types of their components. For example, an expression $(x * y)$ can be assigned type *Int*, if the operands x and y both have type *Int*. Formal specifications of type-systems are usually given by a collection of Gentzen-style inference rules (Gentzen, 1935) like the following:

$$\frac{e_1 :: \mathit{Int} \qquad e_2 :: \mathit{Int}}{e_1 + e_2 :: \mathit{Int}}\ \textsc{Add}$$

Add is a name that allows us to refer to the rule.[1] The judgments above the line are referred to as the *premises* of the rule and the judgment below the line is the *conclusion* of the rule. Notation $e :: \sigma$ can be read as *"expression e has type σ"*. Hence, the above rule can be read as *"if expression e_1 has type Int and expression e_2 has type Int then expression $e_1 + e_2$ has type Int"*.

The main motivation for a programming language designer to employ a type system is for preventing operations from being used with values for which their behavior is unspecified, that is, *logical errors*. For example, the expression $(42 + \text{'}c\text{'})$ is—according to most languages–*ill-typed*, since only numeric values can be added.

In essence, a type system specifies the—otherwise implicit—categories the programmer uses to capture data structures and enforces properties that software artefacts are expected to have, aiming to reduce the number of logical errors or *bugs*.

1.1.1 Dynamic vs. Static Typing

Ensuring that a program is *well-typed* (with respect to the type system specification of the language it is written in) can happen *statically* (at compile-time), *dynamically* (at run-time), or as a combination of both static and dynamic checks.

Static Type Checking Programming languages with static type checking verify the *type-safety*—that is, the absence of run-time errors—of a program entirely at compile-time, before the program runs. For example, ML's type checker would reject function `g`

```
let g(x) = 'c' + x;;
```

[1]Rule names are often omitted if not referred to.

since `'c'` does not have a numeric type and the addition with an integer `x` would lead to a run-time error. Hence, strong static typing can eliminate a large class of bugs at compile-time. Furthermore, once type-checking is complete, compilers for statically-typed languages can often erase all type information before run-time, eliminating the need for run-time type checks that can introduce an overhead.

Dynamic Type Checking On the other hand, programming languages that employ dynamic type checking verify the type-safety of a program at run-time. That is, each object is associated with a run-time representation of its type, which is used to ensure that is only used in well-specified ways. A prime example of a programming language that features a dynamic type system is Python. As an example of dynamic type checking, the corresponding `g` function in Python

```
def g(x): return ('c' + x)
```

would not raise a compiler-time error. Instead, type-errors will arise at run-time, if `g` is actually used. For example, the call `g(1)` results in the following run-time type-error:

```
unsupported operand type(s) for +: ‘int’ and ‘str’
```

Obviously, each kind of typing has its benefits and this is precisely why different programming languages make different design choices on the matter. Static typing is (usually) more restrictive, which can make statically-typed languages cumbersome to use for small, everyday tasks. For example, the expression

```
if true then 5 else 'c'
```

would be rejected by most of the mainstream statically-typed languages, since the two branches of the `if-then-else` clause have different types. Yet, it is not difficult to see that the above expression is no threat to safety; it evaluates to `5`. On the other hand, large-scale projects can easily become difficult to implement, extend, and debug, in which case a static type system can significantly aid the development process.

1.1.2 Static Typing

We mainly focused on one of the benefits of static typing in the previous section: *type safety*. In fact, static typing can be beneficial during software development in numerous ways, the most significant benefits being *safety*, *reusability*, *automated code generation*, and *optimization*.

Safety As we have briefly mentioned already, the most prominent benefit of employing a static type system is to ensure that programs are type-safe. Static type checking can detect software errors (bugs) early in the development process, and thus save development time and reduce debugging and testing costs.

Reusability Moreover, through static typing one can write a single implementation of a language object, e.g., a function, and call it at multiple sites, on arguments of possibly different types. This notion is widely known as *generic programming* (Musser and Stepanov, 1989) and can significantly reduce development time through *code reusability*. Though the same effect can be achieved in the absence of typing, a static type system can also verify that generic functions are used in a type-safe way.

Automated Code Generation Another advantage of static typing is the possibility of *automated code generation*. Compilers can use type information to automatically generate repetitive pieces of code (boilerplate), thus freeing more time for the programmer to focus on more interesting parts of the program. Furthermore, automated code generation further contributes to safety, since it is often based on mathematical foundations and is thus safer than manually-written code.

Optimizations Finally, another benefit of static typing is the opportunities it offers for *code optimization*. If the types of objects handled by a program are known statically (at compile-time), the compiler can specialize function implementations to improve run-time performance, or select efficient run-time representations for complex data structures.

1.1.3 Type-level Features

Though the previous section discussed several benefits of static typing that can significantly decrease development time and increase safety, it is often argued that static typing can also delay the development process. Indeed, dynamically-typed languages offer a lot of freedom to the developer, while statically-typed languages restrict the usage of language constructs. For usability purposes, they compensate for this lack of freedom with an extensive collection of features. We briefly discuss some of the most popular of these features below.

Polymorphism Maybe the most basic and at the same time the most useful extension that has been studied is the notion of *polymorphism*. A function is said to be polymorphic in the type of its argument if it can operate on arguments of multiple types. As an example, a function that reverses a simply-linked list can be polymorphic on the type of the list's elements. This allows one to write a single definition for list reversal, and reuse it on lists containing elements of different types.

Function Overloading Another useful feature that has been extensively studied is that of *function overloading*. A function name is said to be overloaded if it denotes multiple implementations that can be referred to using the same name. One of the most common manifestations of overloading is numeric operators "+", "−", etc., which can be used on integers, floating-point numbers, etc.

Type Invariants and Dependent Types One powerful feature that often appears in formal languages and proof assistants is that of *dependent types*. In this setting, types can refer to terms, thus allowing many program properties to be captured in types that can then be statically verified by the language compiler. For example, one can annotate the type of lists with their (integer) length n. This allows for example the system to verify that a function that reverses a list computes a list of the same length as the input.

In the presence of these more expressive types, the system can also be extended with more features for type-level computation. For example, the concatenation of two length-indexed lists computes a list of length equal to the sum of the input lists. Hence, using such rich types leads to a need for more means for type-level computation, such as type-level functions.

Indeed, programming languages with more expressive type systems usually complement the collection of features they support with features for type-level computation.

1.1.4 The Challenge of Type Reconstruction

One of the most significant challenges of static typing is the ability to omit explicit type-annotations. Though types can aid readability by documenting the meaning of language objects, their size can grow disproportionally (with respect to the size of the program) in languages with expressive static typing. Hence, to address this issue, most language compilers employ *type inference* (or *type reconstruction*), a procedure which statically reconstructs omitted types of language entities.

Unfortunately, the more expressive the type system, the more challenging it is to statically infer types. In fact, type inference for many popular type-level extensions is proven to be undecidable in the total absence of explicit type annotations. Thus, a highly active area of research is concerned with the development of expressive type systems for which types can be inferred by requiring a minimal set of type annotations.

In summary, the urgent need for rich types that can contribute to more accurate modelling and more safety guarantees is undermined by the lack of corresponding type inference methods. Though the problem is known to be hard, there is a lot of room for improvement, either by improving type inference algorithms for existing features, or by developing new type inference algorithms for existing or new features.

1.2 Thesis Overview and Scientific Output

The overall goal of this thesis is the extension of existing and the development of new type-level features, promoting the usage of rich types that can capture and statically enforce program properties.

1.2.1 Aim of the Thesis and Language of Choice

More specifically, in this thesis we extend upon the design of Haskell, a general-purpose functional programming language with strong static typing. Haskell is the ideal starting point for developing novel language extensions, because it has always played a leading role in programming language and type systems research, and constitutes the state-of-the-art in powerful static type systems with type inference in the presence of little or no annotations at all. Moreover, in its position as an influential language it has had considerable impact on the design of many other programming languages, and its developments are closely followed by the programming language community.

In this thesis we improve upon the design of two of the most popular features of Haskell: *pattern matching* and *type classes*. Our aim is to offer more opportunities for static enforcement of program properties, better abstractions for expressive modelling, and more possibilities for automatic code generation.

1.2.2 Part I: Pattern Match Checking

The first part of this thesis is concerned with lazy pattern matching and the techniques we have developed to reason about its properties. More specifically, Part I is split in three chapters (2, 3, and 4). A significant portion of Part I draws its material from the following publication:

> Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis and Simon Peyton Jones (2015). GADTs Meet Their Match: Pattern-matching Warnings That Account for GADTs, Guards, and Laziness. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, pp. 424-436, Vancouver, BC, Canada, August 31-September 2, 2015.

Chapter 2 presents the notion of pattern matching and all its relevant properties, as well as the problem we wish to solve: pattern match checking in the presence of GADTs, guards, and laziness. This problem we address in Chapter 3, where we formalize our pattern match checking algorithm. Finally, Chapter 4 provides technical information concerning the implementation of the algorithm of Chapter 3 in the Glasgow Haskell Compiler.

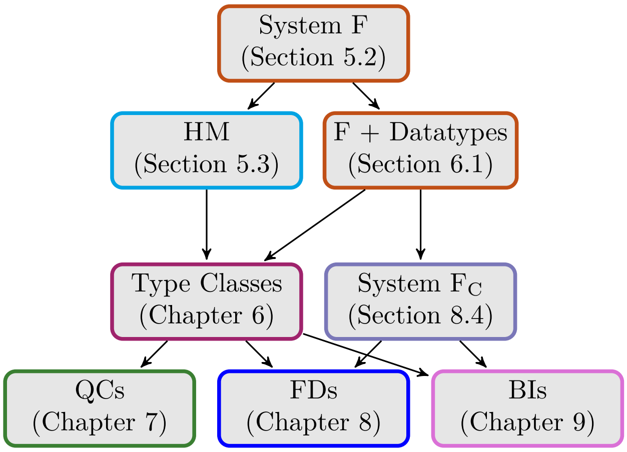
1.2.3 Part II: Type Classes

The overall goal of Part II is to enhance *type classes*, by borrowing ideas from first-order predicate logic. This we achieve by introducing three standalone extensions to the formalization of type classes: *Quantified Class Constraints*, *Functional Dependencies*, and *Bidirectional Type Class Instances.* Each of the aforementioned extensions is inspired by a different aspect of first-order logic.

Background In order to present the new features in a comprehensive way, we develop our extensions in Part II incrementally. This allows us to focus on the feature-specific details and avoid needless repetition. More specifically, Part II presents several calculi, shown in Figure 1.1.

First, Chapter 5 elaborates on the notion of type checking, type inference, type-directed elaboration, and type classes. In the process, it presents two existing calculi: System F (Section 5.2), and the Hindley-Damas-Milner system (HM) (Section 5.3). Next, Chapter 6 develops a formalization of HM with type classes, which serves as the basis for our extensions in the subsequent chapters. Before presenting the specification of this calculus—which we refer to as the "Basic System"—in Section 6.3, we first develop an extension of System

Figure 1.1 Calculi Dependency



F with algebraic data types and recursive let-bindings, given in Section 6.1. The extended System F serves as the target language we elaborate type classes into, using an inference-with-elaboration algorithm presented in Section 6.4.

Due to the non-parametric nature of two of our extensions, Section 8.4 also presents System F_C (Sulzmann et al., 2007a), an extension of System F with GADTs and open, non-parametric type-level functions. Then, we present our extensions in Chapters 7, 8, and 9.

Type Class Extensions For each of the features we present, we develop a complete formalization, including the extensions to (a) the specification of typing and elaboration, (b) the type inference algorithm, (c) the typing-directed elaboration algorithm, and (d) the statement of the most interesting meta-theoretical properties. Since the judgments for each calculus often have the same appearance, we distinguish between them by marking judgments related to each calculus with the corresponding colors in Figure 1.1.

Extension 1: Quantified Class Constraints The first extension, *Quantified Class Constraints*, is presented in Chapter 7. Quantified constraints raise the expressive power of type classes to a subset of first-order logic known as the universal fragment of Hereditary Harrop logic (Harrop, 1956). The contents of Chapter 7 have been published in the following article:

> Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira and Philip Wadler (2017). Quantified Class Constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell '17, pp. 148–161, Oxford, UK, September 7–8, 2017.

From the user's point of view, the merit of quantified constraints manifests itself in terms of more expressive modelling and in terms of terminating type inference for a bigger class of programs.

Extension 2: Functional Dependencies The second extension, *Functional Dependencies*, is presented in Chapter 8. Functional Dependencies have partially been formalized in the literature but we are the first to develop a complete formalization for them, including their elaboration into a statically-typed core language. According to our formalization, functional dependencies extend Haskell types with type-level functions and the logic of constraints with type equalities. The contents of Chapter 8 have been published in the following article:

> Georgios Karachalias and Tom Schrijvers (2017). Elaboration on Functional Dependencies: Functional Dependencies Are Dead, Long Live Functional Dependencies! In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell '17, pp. 133–147, Oxford, UK, September 7–8, 2017.

Functional dependencies significantly raise the expressive power of type classes, enabling type-level computation and more accurate modelling.

Extension 3: Bidirectional Instances The third extension, *Bidirectional Instances*, constitutes part of ongoing work and is presented in Chapter 9. This feature extends type classes' expressive power by elaborating type class instances bidirectionally, effectively adding the biconditional connective to their logical interpretation.

This feature taps into the semantics of type classes, improving upon their interaction with a multitude of type-level features; type-level functions and GADTs can be combined with the bidirectional interpretation of type classes to model strictly more programs than in plain Haskell.

1.2.4 Other Publications

Finally, related to the contents of this thesis but not included, is the following publication:

> Amr Hany Saleh, Georgios Karachalias, Matija Pretnar, and Tom Schrijvers (2018). Explicit Effect Subtyping. In *Proceedings of the 27th European Symposium on Programming*, ESOP '18, Thessaloniki, Greece, April 16–19, 2018.

In this work, we develop an explicitly-typed polymorphic core calculus for algebraic effect handlers with a subtyping-based type-and-effect system. Appeals to subtyping are reified in explicit casts with coercions that witness the subtyping proof, similarly to the type-equality coercions appearing in System $\mathrm{F_C}$. Moreover, our calculus employs an additional namespace, *type skeletons*, that captures the structural part of a type, enabling a straightforward type- and coercion-erasure procedure.

Part I

Pattern Match Checking

Chapter 2

Background

The first part of the thesis is concerned with pattern matching, and particularly reasoning about its properties. In this introductory chapter we present pattern matching, and introduce all relevant notions and terminology that we use in Chapters 3 and 4. More specifically, the structure of this chapter is the following:

Section 2.1 presents algebraic data types and introduces the notion of pattern matching. Section 2.2 elaborates on pattern matching, and introduces the notions of pattern match compilation and checking. Section 2.3 introduces generalized algebraic data types and discusses pattern matching in their presence. Lastly, Section 2.4 elaborates on the problem we aim to solve: reasoning about lazy pattern matching in the presence of GADTs and guards.

2.1 Algebraic Data Types and Pattern Matching

Algebraic Data Types (ADTs) made their first appearance in programming languages in the 1970s as part of Hope (Burstall et al., 1980), a rather small functional language, but quite important in the development of functional programming. Nowadays, they are supported by almost all industrial strength functional languages (e.g., Haskell and OCaml).

Definition An ADT is an ordered pair, consisting of a *type constructor* T together with a set of *data constructors* K. As an example, consider the definition of booleans in Haskell:

$$\textbf{data}\ \mathit{Bool} = \mathit{False} \mid \mathit{True}$$

Type constructor *Bool* captures the new type and data constructors *False* and *True* are the only terms that inhabit type *Bool*.

ADTs can also have *type parameters*, types that are known in the object-oriented world as *generics* (Musser and Stepanov, 1989). A prime example of a generic data type in Haskell is that of singly-linked lists:[1]

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

A term of type $(\mathit{List}\ a)$ is a singly-linked list of elements of type a. For example, $(\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil})))$ has type $(\mathit{List}\ \mathit{Int})$, and represents list $[1, 2, 3]$.

Types a and $(\mathit{List}\ a)$ in the definition of *Cons* are the types of its term-level arguments, which are known as the *fields* of the constructor. Note that in ADT notation for $(T\ \overline{a})$ we omit the return type of the data constructors (which are nothing more than term-level functions), since it is uniquely determined. For example, the types of data constructors *Nil* and *Cons* are:

$$\begin{array}{lll} \mathit{Nil} & :: & \forall a.\ \mathit{List}\ a \\ \mathit{Cons} & :: & \forall a.\ a \to \mathit{List}\ a \to \mathit{List}\ a \end{array}$$

The universal quantifier $\forall$ in both types indicates that they are *parametric* over type parameter a (i.e., they behave uniformly on values of any type). In short, the shape of a term of type $(\mathit{List}\ a)$ is independent of the instantiation of type parameter a. As we illustrate below in Section 2.3, this is not the case for *Generalized ADTs*. We discuss parametricity in more detail in Section 5.1.

Pattern Matching Just like we can construct expressions of type $(T\ \overline{a})$ by applying one of T's data constructors to appropriate arguments, we can also deconstruct a term of type $(T\ \overline{a})$ by means of *pattern matching*. As an example, consider function *reverse*, which reverses a list:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{reverse} :: \forall a.\ [a] \to [a]} \\ \mathit{reverse}\ [] & = & [] \\ \mathit{reverse}\ (x : \mathit{xs}) & = & \mathit{reverse}\ \mathit{xs} \mathbin{+\!\!+} [x] \end{array}$$

[1] Throughout the rest of the thesis we use the built-in syntax of Haskell lists, where *List* is a mixfix operator [_] and *Cons* is an infix operator (_ : _):

$$\textbf{data}\ [a] = [] \mid a : [a]$$

Function *reverse* is defined by means of pattern matching on its argument of type $[a]$. Each line is called a *clause*: a clause consists of a left-hand side which comprises *patterns* to be matched against, and a right-hand side which is to be evaluated if matching against the patterns is successful.

In *reverse*, the first clause covers cases where the argument is an empty list $[]$, and the second covers cases where the argument is constructed by means of constructor $(:)$. Function $(+\!\!+)$ performs standard list concatenation.

Algebraic data types are *closed*, in the sense that the only way to create a term of type $(T\ \overline{a})$ is by means of its data constructors. Hence, *reverse* above covers all possible shapes a list can have.

The benefit of pattern matching is twofold:

1. *Discrimination.* A value of type $[a]$ can be constructed by any of the two data constructors of the list definition. Pattern matching allows a function to behave differently depending on the constructors used in the construction of the *scrutinee* (i.e., the expression being matched against the patterns).
2. *Deconstruction.* Pattern matching also binds subterms (data constructor fields) to fresh term variables (e.g. x and xs in the definition of *reverse*), and brings them in scope for the right-hand side.

Note that in ML-style pattern matching (e.g., in Haskell or OCaml), patterns are *linear*: each variable can appear in a pattern only once. In this thesis, we are only concerned with linear patterns.

There are several important aspects of pattern matching, which we elaborate on in the next section.

2.2 Pattern Matching

We now focus on pattern matching and all its aspects that are relevant to this thesis. More specifically, compilation of pattern matching is discussed in Section 2.2.1, and reasoning about pattern matching in Section 2.2.2.

2.2.1 Compilation of Pattern Matching

It is common for functional programming languages to offer multiple term-level pattern matching constructs. Each is more appropriate for different applications but they can all be elaborated into a single kind of pattern matching construct: *case expressions.* The process of elaborating complex forms of pattern matching into simple case expressions is known as *compilation of pattern matching* and it is the subject of this section.

Case Expressions The most primitive form of pattern matching is that of a *case expression.* For example, *reverse*, which was given by two clauses, each one matching against the function argument, can alternatively be written using a case expression as follows:

$$\begin{array}{l} \mathit{reverse}\ \mathit{xss} = \textbf{case}\ \mathit{xss}\ \textbf{of} \\ \qquad \begin{array}{lcl} [] & \to & [] \\ (x : \mathit{xs}) & \to & \mathit{reverse}\ \mathit{xs} \mathbin{+\!\!+} [x] \end{array} \end{array}$$

A case expression behaves similarly to separate function clauses but it allows matching only a single expression (the scrutinee) against one or more patterns.

Nested Patterns Another feature of pattern matching is the ability to match against the so-called *nested patterns.* As an example, consider the following function which returns the second element of a list:

$$\begin{array}{l} \mathit{second} :: \forall a.\ [a] \to a \\ \mathit{second}\ \mathit{xss} = \textbf{case}\ \mathit{xss}\ \textbf{of} \\ \qquad (x : (y : \mathit{ys})) \ \to\ y \end{array}$$

In this case, the tail of the list pattern is not a simple variable (as was xs in the definition of *reverse*). Instead, it is another list pattern $(y : \mathit{ys})$. Nested patterns allow the binding of arbitrarily nested subterms to variables using pattern matching.

Pattern Matching with Multiple Arguments Function clauses can also match against multiple arguments at once, a possibility not directly offered by case expressions. For example, function *zip* below uses pattern matching against two arguments to combine two lists into a list of tuples (element-wise):

$$\begin{array}{l} \mathit{zip} :: \forall a.\ \forall b.\ [a] \to [b] \to [(a,b)] \\ \begin{array}{llllll} \mathit{zip} & [] & [] & = & [] \\ \mathit{zip} & (a : \mathit{as}) & (b : \mathit{bs}) & = & (a,b) : \mathit{zip}\ \mathit{as}\ \mathit{bs} \end{array} \end{array}$$

Compilation of Pattern Matching All forms of pattern matching we have discussed until now can be elaborated into a simple form with only nested case expressions, where clauses consist only of non-nested patterns.

Such a form is much easier to reason about and makes the transition from a full-blown functional programming language to bytecode simpler. As an example, consider function *zip*, which can be compiled into the following form:

$$\begin{array}{l}
\mathit{zip}\ \mathit{xss}\ \mathit{yss} = \textbf{case}\ \mathit{xss}\ \textbf{of} \\
\qquad [] \quad \to \textbf{case}\ \mathit{yss}\ \textbf{of} \\
\qquad\qquad\qquad [] \quad \to [] \\
\qquad (x : \mathit{xs}) \to \textbf{case}\ \mathit{yss}\ \textbf{of} \\
\qquad\qquad\qquad (y : \mathit{ys}) \to (x, y) : \mathit{zip}\ \mathit{xs}\ \mathit{ys}
\end{array}$$

Nested pattern matches of the above form resemble tree-like automata and are known in the literature as *decision trees* (Maranget and Para, 1994).[2]

Evaluation Strategy An important factor for the compilation of pattern matching is the evaluation strategy of the language at hand. In a strict language (like ML), the scrutinee of a match is fully evaluated before pattern matching is performed. In a lazy language (like Haskell), the scrutinee is evaluated as much as needed for pattern matching to proceed. In Haskell specifically, pattern matching is performed top-to-bottom, left-to-right.

To illustrate why the evaluation-order matters, consider function f:

$$\begin{array}{llll}
f & \_ & \mathit{False} & = 1 \\
f & \mathit{True} & \mathit{True} & = 2 \\
f & \mathit{False} & \mathit{True} & = 3
\end{array}$$

Functions f_1 and f_2 are two (out of many) possible ways we can compile f:

$$\begin{array}{l}
f_1\ x\ y = \textbf{case}\ x\ \textbf{of} \\
\quad \mathit{True} \to \textbf{case}\ y\ \textbf{of}\ \{\ \mathit{True} \to 2;\ \mathit{False} \to 1\ \} \\
\quad \mathit{False} \to \textbf{case}\ y\ \textbf{of}\ \{\ \mathit{True} \to 3;\ \mathit{False} \to 1\ \} \\
\\
f_2\ x\ y = \textbf{case}\ y\ \textbf{of} \\
\quad \mathit{True} \to \textbf{case}\ x\ \textbf{of}\ \{\ \mathit{True} \to 2;\ \mathit{False} \to 3\ \} \\
\quad \mathit{False} \to 1
\end{array}$$

f_1 matches against x first, while f_2 matches against y first. In a language with strict semantics like ML, f_1 and f_2 have the same semantics (f_2 can also be considered more performant, since it performs fewer run-time checks). In a lazy language like Haskell though, we can easily discriminate between the two: take

[2]Another target language that is often used for compilation of pattern matching is that of *backtracking automata* (Maranget and Para, 1994). Yet, the technique used in the major Haskell implementations (notably GHC) targets decision trees. Hence, we refer the reader to the work of Maranget and Para (1994) for more details concerning backtracking automata.

for example call $(f\ \bot\ \mathit{False})$. The call $(f_1\ \bot\ \mathit{False})$ diverges, while $(f_2\ \bot\ \mathit{False})$ returns 1: f_2 is the right compilation of f with respect to the Haskell semantics. Thus, pattern matching in lazy languages introduces one more effect in addition to the two (discrimination and deconstruction) we discussed in Section 2.1: *evaluation*.

Column-based Compilation It is not our goal to give a detailed account of lazy pattern match compilation here; this is a well-studied problem and several algorithms have been introduced to address it (Augustsson, 1985; Wadler, 1987a; Maranget, 1992; Le Fessant and Maranget, 2001, etc.).

Nevertheless, it is worth mentioning that for pattern matching to be compiled efficiently, most algorithms perform a column-based traversal of the clauses of a match. This allows potential grouping of cases, and avoidance of matching against the same expression twice (see for example function f_1 above, which combines the cases where x is *True* into one clause).

This approach—though crucial for the performance of the compiled match—deviates significantly from the semantics of source-level pattern matching, which are usually given in a top-to-bottom, left-to-right manner.

2.2.2 Pattern Matching Anomalies

The Anomalies Given that pattern matching essentially performs a case-by-case analysis, there are two anomalies that can naturally arise: (a) some cases may not be covered, and (b) some cases may be completely subsumed by others. Therefore, the notions of *missing* and *redundant* clauses naturally arise:

Missing clauses. Pattern matching of a sequence of clauses is *exhaustive* if every well-typed argument vector matches one of the clauses. For example, function *zip* from Section 2.2.1

$$\begin{array}{llll} \mathit{zip} & [] & [] & = [] \\ \mathit{zip} & (a : \mathit{as}) & (b : \mathit{bs}) & = (a, b) : \mathit{zip}\ \mathit{as}\ \mathit{bs} \end{array}$$

is not exhaustive because there is a well-typed call that does not match any of its clauses (take for example $(\mathit{zip}\ []\ [\mathit{True}])$). Hence, a clause of the form $\mathit{zip}\ []\ (b : \mathit{bs}) = e$ is *missing*.

Redundant clauses. If there is no well-typed value that matches the left-hand side of a clause, the right-hand side of the clause can never be accessed and the clause is *redundant*. For example, repeating this equation would be redundant:

$$zip\ \ (x : xs)\ \ (y : ys)\ \ =\ \ \ldots$$

Since the application of a partial function to a value outside its domain results in a run-time error, the presence of non-exhaustive pattern matches often indicates a programmer error. Similarly, having redundant clauses in a match is almost never intentional and indicates a programmer error as well.

Good compilers report missing patterns, to warn the programmer that the function is only partially defined. They also warn about completely-overlapped, and hence redundant, equations. Although technically optional for soundness, these warnings are incredibly useful in practice, especially when the program is refactored (i.e., throughout its active life), with constructors being added and removed from the data type.

Detecting the Anomalies The detection of the aforementioned anomalies can be performed at compile-time and is a key asset during software development and refactoring. There are numerous *coverage checking* algorithms which we discuss in detail in Section 3.5, but for a more formal specification of the anomalies and their detection we now briefly discuss some aspects of the work of Maranget (2007).

Firstly, we need to formally specify what it means for a value v to match against a pattern p; in these cases we say that the value is an *instance* of the pattern. More specifically, Maranget (2007) defines the instance relation as follows:[3]

Definition 1 (Instance Relation). *Given any pattern p and a value v such that p and v are of a common type, the instance relation $p \preceq v$ is defined as follows:*

$$\begin{array}{rcll} x & \preceq & v & \\ K\ p_1\ \ldots\ p_n & \preceq & K\ v_1\ \ldots\ v_n & \quad \textit{iff}\ p_i \preceq v_i \quad (i \in [1 \ldots n]) \end{array}$$

In short, a value v is an instance of a pattern p when there exists a substitution θ (which maps all variables in p to values), such that $\theta(p) = v$.

The relation can be easily generalized to pattern vectors, which capture the left-hand side of clauses:

$$p_1 \ldots p_n \preceq v_1 \ldots v_n \qquad \text{iff}\ p_i \preceq v_i \quad (i \in [1 \ldots n])$$

In conventional pattern matching, a pattern match can be viewed as a pattern matrix P, where each row of the matrix represents a clause.

[3]Maranget (2007) also covers *or-patterns* but we omit them here for simplicity.

Thus, a clause $(v_1 \; \dots \; v_n)$ is specified as missing from a pattern matrix P if the following holds

$$((p_{i1} \; \dots \; p_{in}) \not\preceq (v_1 \; \dots \; v_n)) \qquad \forall (p_{i1} \; \dots \; p_{in}) \in P$$

That is, a clause is missing if it is not an instance of any of the clauses. Of course, if no such value vector exists, then the match is deemed exhaustive.

Similarly, for any clause $(p_{i1} \; \dots \; p_{in})$, if every value vector $(v_1 \; \dots \; v_n)$ that is an instance of it is also an instance of a previous clause then the clause is considered redundant (all cases it covers are covered by previous clauses).

Though these definitions are precise for structural pattern matching and have been put to good use by existing formalizations of compilation and coverage checking algorithms (e.g., Maranget and Para (1994); Maranget (2007)), they do not take into account modern pattern matching extensions. This issue is addressed in the remainder of Part I.

2.3 Generalized Algebraic Data Types

Generalized Algebraic Data Types[4] (GADTs) are a generalization of ADTs where data constructors can bind arbitrary existential type variables and constraints (e.g., class or type equality constraints). In recent years, they have appeared in many programming languages, including Haskell (Peyton Jones et al., 2006), OCaml (Garrigue and Normand, 2011) and Ωmega (Sheard, 2004).

As an example of a GADT, consider *length-indexed vectors*:

$$\begin{array}{ll}
\begin{array}{l}
\mathbf{data}\ \mathit{Nat} :: \star\ \mathbf{where} \\
\quad \begin{array}{lcl} \mathit{Zero} & :: & \mathit{Nat} \\ \mathit{Succ} & :: & \mathit{Nat} \to \mathit{Nat} \end{array}
\end{array}
&
\begin{array}{l}
\mathbf{data}\ \mathit{Vec} :: \mathit{Nat} \to \star \to \star\ \mathbf{where} \\
\quad \begin{array}{lcl} \mathit{VN} & :: & \mathit{Vec}\ \mathit{Zero}\ a \\ \mathit{VC} & :: & a \to \mathit{Vec}\ n\ a \to \mathit{Vec}\ (\mathit{Succ}\ n)\ a \end{array}
\end{array}
\end{array}$$

On the left, we define type-level natural numbers *Nat*. Type *Nat* is automatically *promoted* into a kind and data constructors *Zero* and *Succ* into type constructors of the same name, using the GHC extension `DataKinds` (Yorgey et al., 2012).

Length-indexed vectors *Vec* utilize *Nat* to index data constructors *VN* and *VC* with the appropriate length: *VN* represents the empty vector (and thus has length *Zero*), and *VC* represents concatenation (and thus constructs vectors of length *Succ n*, where *n* is the length of the sublist). GADTs allow for

[4]Also known as recursive data types (Xi et al., 2003) and first-class phantom types (Cheney and Hinze, 2003).

more expressive types and stronger static enforcement of program properties. Consider function *vzip*:

$$\begin{array}{llll} \mathit{vzip} :: \mathit{Vec}\ n\ a \to \mathit{Vec}\ n\ b \to \mathit{Vec}\ n\ (a, b) \\ \mathit{vzip}\ \ \mathit{VN} & \mathit{VN} & = & \mathit{VN} \\ \mathit{vzip}\ \ (\mathit{VC}\ x\ \mathit{xs}) & (\mathit{VC}\ y\ \mathit{ys}) & = & \mathit{VC}\ (x, y)\ (\mathit{vzip}\ \mathit{xs}\ \mathit{ys}) \end{array}$$

Though the definition of *vzip* is the same as that of *zip* we presented earlier (Section 2.2), its type is much richer: the fact that the length of the input vectors is the same (and so is the length of the result) is captured in the type of *vzip*: all types are indexed by the same length n.

Existentials and Local Constraints As we mentioned earlier, GADTs are a special case of something more general: *data constructors that bind arbitrary existential type variables and constraints.* That is, the above definition of *Vec* can be equivalently written as:[5]

$$\begin{array}{llll} \textbf{data}\ \mathit{Vec}\ (n :: \mathit{Nat})\ (a :: \star)\ \textbf{where} \\ \quad \mathit{VN} :: (n \sim \mathit{Zero}) & \Rightarrow & \mathit{Vec}\ n\ a \\ \quad \mathit{VC} :: (n \sim \mathit{Succ}\ n') & \Rightarrow & a \to \mathit{Vec}\ n'\ a \to \mathit{Vec}\ n\ a \end{array}$$

This syntax makes it clear that both *VN* and *VC* construct terms of type *Vec n a*, but each one refines index n differently, via *local constraints* ($(n \sim \mathit{Zero})$ for VN and $(n \sim \mathit{Succ}\ n')$ for *VC*).

Such constraints must be satisfied when constructing a term using *VN* or *VC*, and consequently are provided when we pattern match against a term of type *Vec n a*. In the terminology of Vytiniotis et al. (2011), local constraints are *wanted* in the former case and *given* in the latter.

Finally, notice the existentially-quantified variable n' which appears in local constraint $(n \sim \mathit{Succ}\ n')$, as well as pattern argument type $\mathit{Vec}\ n'\ a$: variable n' does not appear in result type *Vec n a*. Both existentially-quantified variables and local constraints complicate reasoning about pattern matching, as we discuss next.

2.4 Problem Statement

The question of determining exhaustiveness and redundancy of pattern matching has been well studied, but almost exclusively in the context of purely structural

[5] Following the Haskell convention, we use infix operator "$\sim$" to denote type equality.

matching (see for example the work of Maranget (2007)). Haskell has moved well beyond simple constructor patterns though: it has overloaded literal patterns, guards, view patterns, pattern synonyms, GADTs, etc. In the remainder of this chapter we illustrate how these features complicate the task of detecting pattern matching anomalies (inexhaustiveness and redundancy). In particular, we identify three new challenges:

- The challenge of GADTs and, more generally, of patterns that bind arbitrary existential type variables and constraints (Section 2.4.1).
- The challenge of guards and guard-like features (Section 2.4.2).
- The challenge of laziness (Section 2.4.3).

These issues are all addressed individually in the literature but to our knowledge we are the first to tackle all three in a single unified framework (Chapter 3), and implement the unified algorithm in a production compiler (Chapter 4).

2.4.1 The Challenge of GADTs

Apart from the well-studied difficulties they pose for type inference (Schrijvers et al., 2009; Vytiniotis et al., 2011), GADTs also introduce a qualitatively-new element to the task of determining missing or redundant patterns. Consider for example function *vzip* from Section 2.3:

$$
\begin{array}{llll}
\mathit{vzip} :: \mathit{Vec}\ n\ a \to \mathit{Vec}\ n\ b \to \mathit{Vec}\ n\ (a,b) \\
\mathit{vzip}\ \ \mathit{VN} & \mathit{VN} & = & \mathit{VN} \\
\mathit{vzip}\ \ (\mathit{VC}\ x\ \mathit{xs}) & (\mathit{VC}\ y\ \mathit{ys}) & = & \mathit{VC}\ (x,y)\ (\mathit{vzip}\ \mathit{xs}\ \mathit{ys})
\end{array}
$$

In contrast to function *zip* of Section 2.2, *vzip is* exhaustive; a call with arguments of unequal length is simply ill-typed. As this indicates, only a *type-aware* algorithm can correctly decide whether or not the pattern matches of a function definition are exhaustive.

Indeed, although GADTs have been supported by the *Glasgow Haskell Compiler* (GHC) since March 2006 (Peyton Jones et al., 2006), the pattern match check was never extended to take GADTs into account, resulting in many user bug reports. Although there have been attempts to improve the algorithm (see tickets[6] #366 and #2006), all were essentially *ad-hoc* and handled only specific cases.

[6]Tickets are GHC bug reports, recorded through the project's bug/issue tracking system: `ghc.haskell.org/trac/ghc`.

2.4.2 The Challenge of Guards

Another pattern matching feature that poses difficulties for the detection of missing and redundant clauses is that of *guards*. Haskell guards come in all shapes and sizes; we discuss their most commonly used forms below.

Boolean Guards Consider this function which computes the absolute value of an integer:

$$\begin{array}{lll} \mathit{abs_1} :: \mathit{Int} \to \mathit{Int} & & \\ \mathit{abs_1}\ x \mid x < 0 & = & -x \\ \qquad\quad \mid \mathit{otherwise} & = & x \end{array}$$

Function $\mathit{abs_1}$ makes use of Haskell's boolean-valued *guards*, introduced by "$|$". If the guard returns *True*, the clause succeeds and the right-hand side is evaluated; otherwise pattern-matching continues with the next clause.

It is clear to the reader that this function is exhaustive, but not so clear to a compiler. Notably, *otherwise* is not a keyword; it is simply a value defined by $\mathit{otherwise} = \mathit{True}$. The compiler needs to know that fact to prove that the pattern-matching is exhaustive. But, what about this version?

$$\begin{array}{lll} \mathit{abs_2} :: \mathit{Int} \to \mathit{Int} & & \\ \mathit{abs_2}\ x \mid x < 0 & = & -x \\ \qquad\quad \mid x \geq 0 & = & x \end{array}$$

Here the exhaustiveness of pattern-matching depends on knowledge of the properties of $<$ and $\geq$. In general, the exhaustiveness for pattern matches involving guards is clearly undecidable; for example, it could depend on a deep theorem of arithmetic. Nevertheless, we would like the compiler to do a good job in common cases such as $\mathit{abs_1}$, and perhaps $\mathit{abs_2}$.

Pattern Guards GHC extends guards further with *pattern guards* (Erwig and Peyton Jones, 2000). For example:

$$\begin{array}{llll} \mathit{append}\ \mathit{xs}\ \mathit{ys} & & & \\ \quad \mid [\,] & \leftarrow \mathit{xs} & = & \mathit{ys} \\ \quad \mid (p : \mathit{ps}) & \leftarrow \mathit{xs} & = & p : \mathit{append}\ \mathit{ps}\ \mathit{ys} \end{array}$$

The pattern guard matches a specified expression (here xs in both cases) against a pattern; if matching succeeds, the guard succeeds, otherwise pattern matching drops through to the next clause.

Let Bindings Another form of guard is that of let-bindings. In contrast to pattern guards, matching in let-bindings is lazy. As an example, consider functions f and g below:

$$\begin{array}{llll} f :: a \to \mathit{Maybe}\ a \to a & & g :: a \to \mathit{Maybe}\ a \to a & \\ f\ x\ y\ \mid \mathit{Just}\ y' \leftarrow y\ =\ y' & & g\ x\ y\ \mid \mathbf{let}\ \mathit{Just}\ y' = y\ =\ y' & \\ \qquad\ \mid \mathit{Nothing} \leftarrow y\ =\ x & & \qquad\ \mid \mathbf{let}\ \mathit{Nothing} = y\ =\ x & \end{array}$$

Function f takes a value of type a (default) and a value of type $\mathit{Maybe}\ a$. If the second value is of the form $\mathit{Just}\ y'$ then y' is returned; otherwise the default value is returned.

The syntax of function g is identical, but instead of pattern guards, it uses let-bindings. Calls $(f\ 42\ (\mathit{Just}\ 1))$ and $(g\ 42\ (\mathit{Just}\ 1))$ behave identically (they both return 1); the difference in the behavior of f and g becomes apparent if we pass *Nothing* as the second argument:

```
Guards> f 42 Nothing
42

Guards> g 42 Nothing
*** Exception: Irrefutable pattern failed for pattern Just y'
```

In the first call, *Nothing* is matched against $\mathit{Just}\ y'$. Matching fails so the second guard is tried. Since this match is successful, the default value 42 is returned.

The situation for g is quite different: a let-binding in a guard position does not actually perform pattern matching. Instead, it binds the result of matching to new pattern variables. In short, the first clause of g above is equivalent to the following:[7]

$$\begin{array}{l} g\ x\ y\ \mid\ y' \leftarrow \mathit{fromJust}\ y\ =\ y' \\ \qquad\quad \mid\ \ldots \end{array}$$

which does not force the evaluation of y. Hence, the first clause matches independently of the value of y and the result is equivalent to $\mathit{fromJust}\ y$.

View Patterns View patterns (Wadler, 1987b; Erwig and Peyton Jones, 2000) are a guard-like extension of pattern matching which allows matching against the result of function applications directly. For example, we can use view patterns to

[7]Where function $\mathit{fromJust} :: \mathit{Maybe}\ a \to a$ unsafely extracts a value from a term of type $\mathit{Maybe}\ a$. For details see library `Data.Maybe`.

define a variant of *find* which takes a default element as an additional argument:

$$\begin{array}{lcl} \mathit{find}' :: a \to (a \to \mathit{Bool}) \to [a] \to a & & \\ \mathit{find}'\ \mathit{def}\ p\ (\mathit{find}\ p \to \mathit{Just}\ x) & = & x \\ \mathit{find}'\ \mathit{def}\ p\ \_ & = & \mathit{def} \end{array}$$

where function $(\mathit{find} :: (a \to \mathit{Bool}) \to [a] \to \mathit{Maybe}\ a)$ returns the first element x of the list for which $p\ x$ is *True*.

The view pattern $(\mathit{find}\ p \to \mathit{Just}\ x)$ is matched successfully against an argument e if $(\mathit{find}\ p\ e)$ matches against $(\mathit{Just}\ x)$. View patterns make reasoning about pattern matching quite challenging since they introduce (a) arbitrary expressions in pattern matching (e.g., function application $(\mathit{find}\ p\ e)$), and (b) non-linear patterns: the second argument (p) is used in the subsequent pattern $(\mathit{find}\ p \to \mathit{Just}\ x)$. The above patterns are not strictly speaking non-linear, but clearly there is a left-to-right dependency between the patterns.

Another interesting aspect of view patterns is that they can appear in arbitrarily nested positions within a pattern. For example, one can write function *map*—which applies a function to all elements of a list—using view patterns as follows:

$$\begin{array}{lcl} \mathit{map} :: (a \to b) \to [a] \to [b] & & \\ \mathit{map}\ f\ [] & = & [] \\ \mathit{map}\ f\ (x : \mathit{map}\ f \to \mathit{xs}) & = & f\ x : \mathit{xs} \end{array}$$

In summary, view patterns offer great flexibility to programmers but introduce several challenges to pattern match checking. Indeed, we know of no coverage algorithm that can give accurate warnings in the presence of view patterns and guards.

Literal Patterns Another known challenge of pattern matching in the ML family is that of literal patterns. First, we can have enumerably many literal patterns as part of the same signature. For example, *Int* is inhabited by all numbers in the range

$$-2147483648\ \ldots\ -1\ 0\ 1\ \ldots\ 2147483647$$

In the case of *Integer*s, the problem becomes worse: they are infinitely many. Haskell further complicates matters with *overloaded literals*. For example, in Haskell one can write the following definition:

$$\begin{array}{lcl} h\ (\mathit{Just}\ 4) & = & \mathit{True} \\ h\ 5 & = & \mathit{False} \end{array}$$

h has the type

$$(\mathit{Eq}\ a, \mathit{Num}\ a) \Rightarrow \mathit{Maybe}\ a \to \mathit{Bool}$$

Though this definition seems ill-typed, it can be accepted by the compiler in the presence of a $(\mathit{Num}\ (\mathit{Maybe}\ a))$ instance:

$$\mathbf{instance}\ \mathit{Num}\ a \Rightarrow \mathit{Num}\ (\mathit{Maybe}\ a)$$

The idea behind overloaded literals is that the pattern represents an equality check (hence the $(\mathit{Eq}\ a)$ constraint above). For example, the second clause above checks whether the argument is equal to $(\mathit{fromInteger}\ 5)$.

Though there exists prior work on reasoning about literal patterns (e.g., (Sestoft, 1996)), we know of no work that deals with overloaded literals, or with simple literals in the presence of all the features we have discussed.

Summary All the aforementioned guard and guard-like extensions pose a challenge to determining the exhaustiveness and redundancy of pattern-matching, because pattern matching is no longer purely structural. Every real compiler must grapple with this issue, but no published work gives a systematic account of how to do so. This is the challenge we address in Chapter 3.

2.4.3 The Challenge of Laziness

Haskell is a lazy language, and it turns out that laziness interacts in an unexpectedly subtle way with pattern matching checks. Here is an example, involving two GADTs:

$$\begin{array}{lll}
\mathbf{data}\ F\ a\ \mathbf{where} & \mathbf{data}\ G\ a\ \mathbf{where} & h :: F\ a \to G\ a \to \mathit{Int} \\
\quad F_1 :: F\ \mathit{Int} & \quad G_1 :: G\ \mathit{Int} & h\ F_1\ G_1 = 1 \\
\quad F_2 :: F\ \mathit{Bool} & \quad G_2 :: G\ \mathit{Char} & h\ \_\ \ \_\ \ = 2
\end{array}$$

Given h's type signature, its only well-typed non-bottom arguments are F_1 and G_1 respectively. So, is the second clause for h redundant? No. Consider the call $(h\ F_2\ \bot)$, where $\bot$ is a diverging value. Pattern matching in Haskell works top-to-bottom, and left-to-right. So we try the first equation, and match the pattern F_1 against the argument F_2. The match fails, so we fall through to the second equation, which succeeds, returning 2.

This subtlety is not restricted to GADTs. Consider:

$$\begin{array}{llll}
\multicolumn{4}{l}{g :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Int}} \\
g & \_ & \mathit{False} & = 1 \\
g & \mathit{True} & \mathit{False} & = 2 \\
g & \_ & \_ & = 3
\end{array}$$

Is the second equation redundant? It certainly *looks* redundant: if the second clause matches, then the first clause would have matched too, so g cannot possibly return 2. The right-hand side of the second clause is certainly dead code.

Surprisingly, though, *it is not correct to remove the second equation.* What does the call ($g\ \bot\ \mathit{True}$) evaluate to, where $\bot$ is a looping value? Answer: the first clause fails to match, so we attempt to match the second. That requires us to evaluate the first argument of the call, $\bot$, which will loop. But if we omitted the second clause, ($g\ \bot\ \mathit{True}$) would return 3.

In short, even though the right-hand side of the second equation is dead code, the equation cannot be removed without (slightly) changing the semantics of the program. As far as we know, this observation has not been made before, although previous work by Maranget (2007) would quite sensibly classify the second equation as non-redundant.

The same kind of thing happens with GADTs. With the same definitions for F and G, consider

$$\begin{array}{l}k :: F\ a \to G\ a \to \mathit{Int}\\ k\ \ F_1\ \ G_1\ \ =\ 1\\ k\ \ \_\ \ \ G_1\ \ =\ 2\end{array}$$

Is the second equation redundant? After all, anything that matches it would certainly have matched the first equation (or caused divergence if the first argument was $\bot$). So the RHS is definitely dead code; k cannot possibly return 2. But removing the second clause would make the definition of k inexhaustive: consider the call ($k\ F_2\ \bot$).

What's more, Haskell extends the language of patterns with two syntactic forms which offer more control over the evaluation order in pattern matching: *strict patterns* and *lazy patterns*. Both syntactic forms further complicate reasoning about pattern matching; we illustrate the delicacies of each form below.

Strict Patterns Strict patterns (also known as *bang patterns*) allow a programmer to explicitly mark a match to have non-lazy semantics. For example, consider function *idl*:

$$\begin{array}{lcl}\mathit{idl}\ [] & = & []\\ \mathit{idl}\ (!x : \mathit{xs}) & = & x : \mathit{idl}\ \mathit{xs}\end{array}$$

Function *idl* performs the identity transformation on lists, but it forces both their spine and their elements to be evaluated, via strict pattern $!x$.

The semantics of strict patterns can be seen if we compare the calls $(\mathit{length}\ (\mathit{idl}\ [\bot,\bot]))$ and $(\mathit{length}\ (\mathit{id}\ [\bot,\bot]))$, where *id* is the standard, lazy identity function. The former diverges, while the latter returns 2.

Notice that a strict pattern does not require the expression matched against it to be fully evaluated; it requires evaluation to WHNF only. Subsequently, annotating a pattern that is already in WHNF with a (!) has no effect (e.g., $(x : xs)$ vs. $!(x : xs)$).

Lazy Patterns Lazy patterns (also known as *irrefutable patterns*) capture the opposite semantics: the argument is not evaluated, even if the pattern is in WHNF. For example, given function

$$\mathit{add}\ ({\sim}\ (x, y)) = x + y$$

matching will succeed, even in the call $(\mathit{add}\ \bot)$ (though the overall result will of course be $\bot$). That is, a lazy pattern matches lazily against the provided argument. Equivalently, one could write function *add* as follows:

$$\begin{array}{l} \mathit{add}\ t = x + y \\ \quad \textbf{where} \\ \qquad x = \mathit{fst}\ t \\ \qquad y = \mathit{snd}\ t \end{array}$$

Summary The bottom line is this: if we want to report accurate warnings, we must take the order of evaluation into account. Plain Haskell is lazy which is a challenge on its own, and features like GADTs, strict patterns, and lazy patterns complicate matters more. We illustrate how to address this challenge in the next chapter.

Chapter 3

Pattern Match Checking

Despite the runaway popularity of GADTs, and other pattern-matching features such as view patterns, boolean guards, pattern guards, overloaded literals, and let-bindings, no production compiler known to us gives accurate pattern-match overlap and exhaustiveness warnings when these features are used. Certainly the state-of-the-art Haskell compiler (GHC) does not. In this chapter we solve this problem by developing a pattern match checking algorithm which takes all the aforementioned features into account.

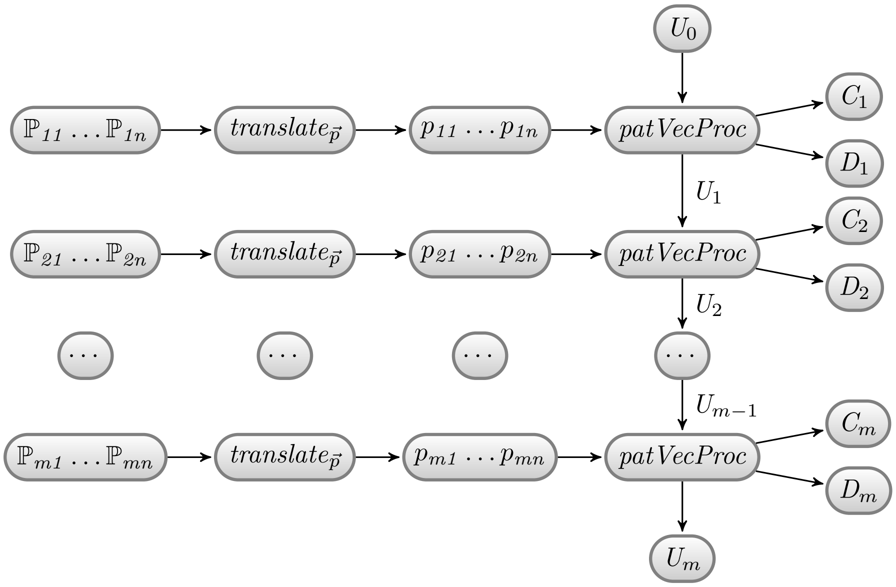
The chapter is structured as follows: Section 3.1 describes our approach in intuitive terms. Next, Sections 3.2 and 3.3 present all formal aspects of our algorithm. Section 3.4 states the most significant meta-theoretical properties of the algorithm of Section 3.3. Section 3.5 discusses a large body of related work, concerned with pattern matching and pattern match checking in particular. Finally, Section 3.6 summarizes the main results presented in the chapter.

3.1 Our Approach

In this section we describe our approach in intuitive terms, showing how it addresses each of the three challenges of Section 2.4. We subsequently formalize the algorithm in Sections 3.2 and 3.3.

Throughout the remainder of Part I we make the assumption that the pattern-match warning pass runs once type inference is complete. At this stage the syntax tree is richly decorated with type information, but has not yet been desugared. Warnings will therefore refer to the program text written by the

Figure 3.1 Algorithm Outline



user, and not some radically-desugared version. Indeed, in our implementation (which we elaborate on in Chapter 4) the pattern match warning generator is triggered just before desugaring each pattern match.

3.1.1 Algorithm Outline

The most common use of pattern matching in Haskell is when a function is defined using multiple *clauses*:

$$\begin{array}{lcll} f\ p_{11}\ \ldots p_{1n} & = & e_1 & \qquad \text{Clause 1} \\ \quad \ldots & = & & \qquad \quad \ldots \\ f\ p_{m1} \ldots p_{mn} & = & e_m & \qquad \text{Clause } m \end{array}$$

From the point of view of pattern matching, the function name "f" is incidental: all pattern matching in Haskell can be regarded as a sequence of clauses, each clause comprising a pattern vector and a right-hand side. For example, a case expression also has multiple clauses (each with only one pattern); a Haskell pattern matching lambda has a single clause (perhaps with multiple patterns); and so on.

As we discussed in Section 2.2.1, in Haskell pattern matching on a sequence of clauses is carried out top-to-bottom, and left-to-right. In our function f above, Haskell matches the first argument against p_{11}, the second against p_{12} and so on. If all n patterns in the first clause match, the right-hand side is chosen; if not, matching resumes with the next clause. Our algorithm, illustrated in Figure 3.1, works in the same way: it analyzes the clauses one by one, from top to bottom. Function $translate_{\vec{p}}$ desugars source patterns into a small yet expressive pattern language; we elaborate on this translation in Section 3.2 below. The analysis *patVecProc* of an individual clause takes a compact symbolic representation of the vector of argument values U_{i-1} that are possibly submitted to the clause, and partitions these values into three different groups:

- C The values that are *covered* by the clause; that is, values that match the clause without divergence, so that the right-hand side is evaluated.
- D The values that possibly *diverge* when matched against the clause, so that the right-hand side is not evaluated, but neither are any subsequent clauses matched.
- U The remaining *uncovered* values; that is, the values that fail to match the clause, without divergence.

As illustrated in Figure 3.1, the input to the first clause represents all possible values, and each subsequent clause is fed the uncovered values of the preceding clause. For example, consider the function *zip* from Section 2.2:

$$\begin{array}{llcl} \mathit{zip} :: [a] \to [b] \to [(a,b)] & & & \\ \mathit{zip}\ [] & [] & = & [] \\ \mathit{zip}\ (a : as) & (b : bs) & = & (a,b) : \mathit{zip}\ as\ bs \end{array}$$

We start the algorithm with $U_0 = \{\_\ \_\}$, where we use “_” to stand for “all values”. Processing the first clause gives:

$$\begin{array}{lcl} C_1 & = & \{[]\ []\} \\ D_1 & = & \{\bot\ \_, []\ \bot\} \\ U_1 & = & \{[]\ (\_ : \_), (\_ : \_)\ \_\} \end{array}$$

The values that fail to match the first clause, and do so without divergence, are U_1, and these values are fed to the second clause. Again we divide the values into three groups:

$$\begin{array}{lcl} C_2 & = & \{(\_ : \_)\ (\_ : \_)\} \\ D_2 & = & \{(\_ : \_)\ \bot\} \\ U_2 & = & \{[]\ (\_ : \_), (\_ : \_)\ []\} \end{array}$$

Now, U_2 describes the values that fail to match either clause. Since it is non-empty, the clauses are not exhaustive, and a warning should be generated. In general we generate three kinds of warnings:

1. If the function is defined by m clauses, and U_m is non-empty, then the clauses are non-exhaustive, and a warning should be reported. It is usually helpful to include the set U_m in the error message, so that the user can see which patterns are not covered.
2. Any clause i for which C_i and D_i are both empty is redundant, and can be removed altogether.
3. Any clause i for which C_i is empty, but D_i is not, has an inaccessible right-hand side even though the equation cannot be removed. This is unusual, and deserves its own special kind of warning; again, including D_i in the error message is likely to be helpful.

Each of C, U, and D is a set of *value abstractions*, a compact representation of a set of value vectors that are covered, uncovered, or diverge respectively. For example, the value abstraction $(\_ : \_)\ []$ stands for value vectors such as

$$\begin{array}{ll} (\mathit{True} : []) & [] \\ (\mathit{False} : (\mathit{True} : [])) & [] \end{array}$$

and so on. Notice in D_1, D_2 that our value abstractions must include $\bot$, so that we can describe values that cause matching to diverge.

3.1.2 Handling Constraints

Next we discuss how these value abstractions may be extended to handle GADTs. Recall *vzip* from Section 2.3:

$$\begin{array}{llll} \multicolumn{4}{l}{\mathit{vzip} :: \mathit{Vec}\ n\ a \to \mathit{Vec}\ n\ b \to \mathit{Vec}\ n\ (a, b)} \\ \mathit{vzip}\ \mathit{VN} & \mathit{VN} & = & \mathit{VN} \\ \mathit{vzip}\ (\mathit{VC}\ x\ \mathit{xs}) & (\mathit{VC}\ y\ \mathit{ys}) & = & \mathit{VC}\ (x, y)\ (\mathit{vzip}\ \mathit{xs}\ \mathit{ys}) \end{array}$$

What do the uncovered sets U_i look like? Naively they would look like that for *zip*:

$$\begin{array}{lll} U_1 & = & \{\mathit{VN}\ (\mathit{VC}\ \_\ \_), (\mathit{VC}\ \_\ \_)\ \_\ \} \\ U_2 & = & \{\mathit{VN}\ (\mathit{VC}\ \_\ \_), (\mathit{VC}\ \_\ \_)\ \mathit{VN}\} \end{array}$$

To account for GADTs we add *type constraints* to our value abstractions, to give this instead:

$$\begin{array}{lllll} U_1 = \{ & \mathit{VN} & (\mathit{VC}\ \_\ \_) & \triangleright & (n \sim \mathit{Zero}, n \sim \mathit{Succ}\ n_2) \\ , & (\mathit{VC}\ \_\ \_) & \_ & \triangleright & (n \sim \mathit{Succ}\ n_2)\ \} \end{array}$$

Each value tuple abstraction in the set now comes with a type equality constraint (e.g. $n \sim \mathit{Succ}\ n_2$), and represents values of the specified syntactic shape, *for* *which the equality constraint is satisfiable* at least for some substitution of its free variables. The first abstraction in U_1 has a constraint that is *unsatisfiable*, because n cannot simultaneously be equal to both *Zero* and *Succ* n_2. Hence the first abstraction in U_1 represents the empty set of values and can be discarded. Discarding it, and processing the second clause gives

$$U_2 \quad = \quad \{(\mathit{VC}\ \_\ \_)\ \mathit{VN} \triangleright (n \sim \mathit{Succ}\ n_3, n \sim \mathit{Zero})\}$$

Again the constraint is unsatisfiable, so U_2 is empty, which in turn means that the function is exhaustive.

We have been a bit sloppy with binders (e.g., where is n_2 bound?), but we will tighten that up in Section 3.3. The key intuition is this: *the abstraction $u \triangleright \Delta$ represents the set of values whose syntactic shape is given by u, and for which the type constraint Δ is satisfied.*

3.1.3 Guards and Oracles

In the previous section we extended value abstractions with a conjunction of type-equality constraints. It is straightforward to take the idea further, and add term-equality constraints. Then the final uncovered set for function abs_2 of Section 2.4.2 might look like this:

$$U_2 \quad = \quad \{x \triangleright (\mathit{False} = x < 0, \mathit{False} = x \geq 0)\}$$

We give the details of how we generate this set in Section 3.3, but intuitively the reasoning goes like this: if neither clause for abs_2 matches, then both boolean guards must evaluate to *False*. Now, if the compiler knows enough about arithmetic, it may be able to determine that the constraint is unsatisfiable, and hence that U_2 is empty, and hence that abs_2 is exhaustive.

For both GADTs and guards, the question becomes this: *is the constraint Δ unsatisfiable?* And that is a question that has been *extremely* well studied, for many particular domains (see for example Zeno (Sonnex et al., 2012) and HipSpec (Claessen et al., 2013)). For the purposes of this chapter, therefore, we treat satisfiability as a black box, or oracle: the algorithm is parameterized over the choice of oracle. This modular separation of concerns is extremely helpful, and is a key contribution of our approach (see Section 3.6).

3.2 Phase 1: Pattern Desugaring

As illustrated in Figure 3.1, before a clause is processed by function *patVecProc*, we translate it into a concise "core" pattern language; this is the language the

Figure 3.2 Source and Target Patterns, Guards, and Clauses

$$\begin{array}{lcll}
f, g, x, y, \ldots & ::= & \langle \textit{term variable} \rangle & \\
K & ::= & \langle \textit{data constructor} \rangle & \\
e & ::= & \ldots & \textit{expression}
\end{array}$$

$$\begin{array}{lcll}
\mathbb{P} & ::= & x \mid \_ \mid K\ \vec{\mathbb{P}} \mid l \mid ol \mid x + l \mid e \to \mathbb{P} \mid !\mathbb{P} \mid \sim \mathbb{P} & \textit{source pattern} \\
\mathbb{G} & ::= & e \mid \mathbb{P} \leftarrow e \mid \textbf{let}\ \mathbb{P} = e & \textit{source guard} \\
\mathbb{C} & ::= & \vec{\mathbb{P}} \mid \vec{\mathbb{G}} \to e & \textit{source clause}
\end{array}$$

$$\begin{array}{lcll}
p, q & ::= & x \mid K\ \vec{p} \mid G & \textit{target pattern} \\
G & ::= & \vec{p} \leftarrow e & \textit{target guard} \\
c & ::= & \vec{p} \to e & \textit{target clause}
\end{array}$$

algorithm operates on. The translation of source patterns into the core language is the focus of this section.

Section 3.2.1 provides the formal syntax of both source and target patterns, and Section 3.2.2 elaborates on the translation from the former to the latter.

3.2.1 Syntax

The syntax of source and target patterns presented in Figure 3.2. We use meta-variables g, f, x, y to denote term variables and K to denote data constructors. By convention, we use f, g to denote functions, and x, y to denote any kind of pattern variable. Expressions are denoted by e and we leave their syntax open. As we illustrate in Section 3.3, our algorithm is indifferent towards the shape of expressions; the checker gives rise only to specific kinds of expressions.

Source Patterns The syntax of source patterns $\mathbb{P}$ is given in Figure 3.2 and captures the most interesting forms of patterns and guards supported by GHC.

A source pattern $\mathbb{P}$ can be of one the following: a term variable x, a *wildcard* pattern _, a constructor pattern $K\ \vec{\mathbb{P}}$, a literal l, an overloaded literal ol, an `n+k` pattern $(x + l)$ (Peyton Jones, 2003), a *view* pattern $e \to \mathbb{P}$ (Wadler, 1987b; Erwig and Peyton Jones, 2000), a *strict* pattern $!\mathbb{P}$, a *lazy* pattern $\sim \mathbb{P}$. The meaning of each pattern is the one we gave in intuitive terms earlier in Sections 2.4.2 and 2.4.3.

A guard can take one of three forms. It can be a boolean guard e, a pattern guard $(\mathbb{P} \leftarrow e)$, or a let-binding $(\textbf{let}\ \mathbb{P} = e)$. Boolean guards have the expected meaning: for matching to proceed, expression e must evaluate to *True*. The difference between the operational semantics of pattern guards and let-bindings is concentrated in the evaluation order they follow, as we discussed in Section 2.4.2: pattern guards are checked as soon as execution meets them, while let-bindings are evaluated *lazily*.

Finally, source clauses $\mathbb{C}$ take the form $\vec{\mathbb{P}} \mid \vec{\mathbb{G}} \to e$, that is, a pattern vector followed by a guard vector. This might seem a bit restrictive at first glance, since standard Haskell allows for multiple guard vectors after the same pattern vector:

$$\begin{aligned} \mathbb{C} ::= \vec{\mathbb{P}} & \mid \vec{\mathbb{G}}_1 \\ & \mid \ldots \\ & \mid \vec{\mathbb{G}}_n \end{aligned}$$

Nevertheless, the two forms are equally expressive. The following n clauses have the same semantics as the clause $\mathbb{C}$ above:

$$\begin{array}{rcl} \mathbb{C}_1 & = & \vec{\mathbb{P}} \mid \vec{\mathbb{G}}_1 \\ & \cdots & \\ \mathbb{C}_n & = & \vec{\mathbb{P}} \mid \vec{\mathbb{G}}_n \end{array}$$

Target Patterns A target *clause* c is a vector of patterns $\vec{p}$ and a right-hand side e, which should be evaluated if the pattern matches. Here, a "vector" $\vec{p}$ of patterns is an ordered sequence of patterns, similarly to a source pattern vector: it is either empty, written ϵ, or is of the form $p\ \vec{p}$.

A pattern p is either a variable pattern x, a constructor pattern $K\ \vec{p}$ or a *guard* G. One difference between source guards $\mathbb{G}$ and target guards G is that the latter match an expression against a pattern vector, not a pattern. Yet, the intended semantics is that the pattern vector $\vec{p}$ consists of exactly one pattern, possibly followed by several guard patterns.

Furthermore, the syntax of target guards greatly generalizes that of source patterns in that it allows a guard to occur *arbitrarily nested inside a pattern*. This allows us to desugar literal patterns and view patterns. For example, consider the Haskell function

$$\begin{array}{lcl} f\ \ (\texttt{'}x\texttt{'}, []) & = & \mathit{True} \\ f\ \ \_ & = & \mathit{False} \end{array}$$

The equality check against the literal character $\texttt{'}x\texttt{'}$ must occur *before* matching the second component of the tuple, so that the call $(f\ (\texttt{'}y\texttt{'}, \bot))$ returns *False*

Figure 3.3 Pattern, Guard, and Clause Desugaring

$\boxed{translate_p :: \mathbb{P} \to \vec{p}}$ Pattern Translation

$$
\begin{array}{lcl}
translate_p(x) & = & x \\
translate_p(\_) & = & y \\
translate_p(K\ \vec{\mathbb{P}}) & = & K\ (translate_{\vec{p}}(\vec{\mathbb{P}})) \\
translate_p(l) & = & x\ (translate_g(x \texttt{ == } l)) \\
translate_p(ol) & = & x\ (translate_g(x \texttt{ == } from\ ol)) \\
translate_p(n+k) & = & x\ (translate_g(x \texttt{ >= } k))\ (n \leftarrow x - k) \\
translate_p(f \to \mathbb{P}) & = & x\ (translate_g(\mathbb{P} \leftarrow f\ x)) \\
translate_p(!\mathbb{P}) & = & x\ (\texttt{()} \leftarrow seq\ x\ \texttt{()})\ (translate_g(\mathbb{P} \leftarrow x)) \\
translate_p(\sim \mathbb{P}) & = & x\ (translate_g(\textbf{let}\ \mathbb{P} = x))
\end{array}
$$

$\boxed{translate_g :: \mathbb{G} \to \vec{p}}$ Guard Translation

$$
\begin{array}{lcll}
translate_g(e) & = & True \leftarrow e & \\
translate_g(\mathbb{P} \leftarrow e) & = & translate_p(\mathbb{P}) \leftarrow e & \\
translate_g(\textbf{let}\ \mathbb{P} = e) & = & \overline{(y \leftarrow (\lambda\mathbb{P}.\ y)\ e)} & (\bar{y} = vars(\mathbb{P}))
\end{array}
$$

$\boxed{translate_c :: \mathbb{C} \to \vec{p}}$ Clause Translation

$$
translate_c(\vec{\mathbb{P}} \,|\, \vec{\mathbb{G}}) \quad = \quad translate_{\vec{p}}(\vec{\mathbb{P}})\ translate_{\vec{g}}(\vec{\mathbb{G}})
$$

rather than diverges. With our syntax we can desugar f to these two clauses:

$$
\begin{array}{lcl}
(y\ (True \leftarrow y\texttt{==}\texttt{'}x\texttt{'}), []) & \to & True \\
z & \to & False
\end{array}
$$

Note the nested guard $True \leftarrow y\texttt{==}\texttt{'}x\texttt{'}$. The complete formal translation of source patterns to target patterns is given in Section 3.2.2 below.

This unusual design choice simplifies the implementation of the algorithm (see Section 3.3.2).

3.2.2 From Source to Target Patterns

The translation from source to target clauses is performed by function $translate_c$. Mutually recursive auxiliary functions $translate_p$ and $translate_g$ translate patterns and guards, respectively. All three are presented in Figure 3.3. We discuss each in detail below.

Pattern Translation Translation of variable patterns x is trivial and wildcards _ are translated into fresh pattern variables, for uniformity. Translation of constructor patterns is straightforward: we translate all arguments and concatenate the results.[1]

Literal patterns l are translated in two steps: first, we generate a fresh term variable x, and second, we add a guard (x == l), to ensure that matching succeeds only if x is equal to l. Overloaded literals ol are treated similarly, but we first call function *from*, to capture the expected semantics. For example, matching an expression e against pattern ($5 :: \mathit{Int}$) corresponds to an equality check between e and ($\mathit{fromInteger}\ 5$).[2]

$n + k$ patterns are translated according to their expected semantics. A value x matches an $n + k$ pattern if x is greater or equal to k, and the difference $x - k$ is bound to variable n.

View patterns $f \to \mathbb{P}$ directly correspond to pattern guards: a view pattern $f \to \mathbb{P}$ is matched by a value x if $f\ x$ matches against pattern $\mathbb{P}$. Fresh variable x captures the actual argument and the matching of $f\ x$ against $\mathbb{P}$ is captured in the additional pattern guard.

In order to translate strict patterns $!\mathbb{P}$ in a semantics-preserving manner, we introduce a call to the built-in function *seq*. Fresh variable x is used to bind the argument, pattern guard $(() \leftarrow \mathit{seq}\ x\ ())$ ensures that the argument is in *Weak Head Normal Form* (WHNF), and guard $(\mathbb{P} \leftarrow x)$ captures the actual match. Notice that the call to *seq* is essential for cases where pattern $\mathbb{P}$ forces no evaluation (e.g., when $\mathbb{P}$ is a variable).

Finally, irrefutable patterns are translated into a (non-forcing) variable x, accompanied by a (lazy) let-binding. This preserves the semantics we discussed in Section 2.4.2.

Guard Translation Guards are translated using function $\mathit{translate}_g$, the definition of which is also given in Figure 3.3.

The first clause handles boolean guards; a boolean guard is satisfied if it evaluates to *True*. For example, the clauses of abs_1 (Section 2.4.2) would desugar to:

$$\begin{array}{lll} x\ (\mathit{True} \leftarrow x < 0) & \to & -x \\ x\ (\mathit{True} \leftarrow \mathit{otherwise}) & \to & x \end{array}$$

[1] A task performed by function $\mathit{translate}_{\vec{p}}$, whose trivial definition we omit.

[2] Where function $\mathit{fromInteger} :: \mathit{Num}\ a \Rightarrow \mathit{Integer} \to a$ is the overloaded method found in the *Num* type class.

The second clause handles pattern guards ($\mathbb{P} \leftarrow e$), by simply translating the pattern $\mathbb{P}$. Worth mentioning is that our initial formalization[3] treated pattern guards slightly differently:

$$\mathit{translate}_g(\mathbb{P} \leftarrow e) \; = \; (p \leftarrow e) \; \vec{p} \qquad \text{where } p \; \vec{p} = \mathit{translate}_p(\mathbb{P})$$

Indeed, function $\mathit{translate}_p$ always (by construction) gives rise to a non-guard pattern followed by a list of guards. This allowed us to keep the shape of target guards similar to that of source pattern guards without loss of expressivity. Nevertheless, this design choice added needless complexity in our implementation and was hence abandoned.

Finally, the third clause handles (lazy) let-bindings. In essence, a lazy let-binding ($\textbf{let}\ \mathbb{P} = e$) always matches, independently of the shape of pattern $\mathbb{P}$: a non-exhaustive let-binding will manifest itself only when it is forced. Thus, each variable y in $\mathbb{P}$ is bound to the corresponding component, without forcing any evaluation.

For example, the following clause

$$f \;\, \mathit{xss} \mid (\textbf{let}\ (x : \mathit{xs}) = \mathit{xss}) = \ldots$$

will be translated as follows:

$$\mathit{xss} \; (x \leftarrow (\lambda(x : \mathit{xs}).\ x) \; \mathit{xss}) \; (\mathit{xs} \leftarrow (\lambda(x : \mathit{xs}).\ \mathit{xs}) \; \mathit{xss})$$

Functions $\mathit{head} \equiv (\lambda(x : \mathit{xs}).\ x)$ and $\mathit{tail} \equiv (\lambda(x : \mathit{xs}).\ \mathit{xs})$ are non-exhaustive, but the let-binding covers all cases.

Clause Translation Last, clause translation is implemented in terms of auxiliary functions $\mathit{translate}_{\vec{p}}$ and $\mathit{translate}_{\vec{g}}$. The former translates all patterns in a pattern vector and concatenates the results and the latter behaves similarly on guard vectors. The translation of a clause ($\vec{\mathbb{P}} \mid \vec{\mathbb{G}}$) is simply the concatenation of the translation of its subparts $\vec{\mathbb{P}}$ and $\vec{\mathbb{G}}$.

3.3 Phase 2: Pattern Match Checking

We now turn to the formal design of our pattern match checking algorithm. First, Section 3.3.1 introduces the additional constructs used by the algorithm; the algorithm itself is explained at length in the remainder of the chapter.

[3]See the extended version of the work of Karachalias et al. (2015): `https://gkaracha.github.io/papers/gadtpm_ext.pdf`.

Figure 3.4 Value Abstractions

$$\begin{array}{lcl} a, b, a', b', \ldots & ::= & \langle \textit{type variable} \rangle \\ T & ::= & \langle \textit{type constructor} \rangle \end{array}$$

$$\begin{array}{lcll} \tau & ::= & a \mid \tau_1 \to \tau_2 \mid T\ \overline{\tau} \mid \ldots & \textit{monotype} \\ \Gamma & ::= & \epsilon \mid \Gamma, a \mid \Gamma, x : \tau & \textit{typing environment} \end{array}$$

$$\begin{array}{lcll} S, C, U, D & ::= & \overline{v} & \textit{value set abstraction} \\ v & ::= & \Gamma \vdash \vec{u} \rhd \Delta & \textit{value vector abstraction} \\ u, w & ::= & x \mid K\ \vec{u} & \textit{value abstraction} \end{array}$$

$$\begin{array}{lcll} \Delta & ::= & \epsilon \mid \Delta \cup \Delta \mid Q \mid x \approx e \mid x \approx \bot & \textit{constraint} \\ Q & ::= & \tau_1 \sim \tau_2 \mid \ldots & \textit{type constraint} \end{array}$$

3.3.1 Value Abstractions

Figure 3.4 extends the syntax of target patterns we gave in Figure 3.2 with the additional constructs we use in the formalization of the algorithm in the next section.

The syntax for monotypes, type constraints and typing environments is entirely standard. We are explicit about the binding of type variables in Γ, but throughout Part I we assume they all have kind $*$, so we omit their kind ascriptions.[4]

Value abstractions play a central role in our work, and stand for sets of values. They come in three forms:

- A *value set abstraction* S is a set of value abstractions $\overline{v}$. We use an overline $\overline{v}$ (rather than an arrow) to indicate that the order of items in S does not matter.
- A *value vector abstraction* v has the form $\Gamma \vdash \vec{u} \rhd \Delta$. It consists of a vector $\vec{u}$ of syntactic value abstractions, and a constraint Δ. The type environment Γ binds the free variables of $\vec{u}$ and Δ.
- A *syntactic value abstraction* u is either a variable x, or is of the form $K\ \vec{u}$, where K is a data constructor.

[4]As we explain in Chapter 4, our implementation supports higher kinds, and indeed kind polymorphism.

A value abstraction represents a set of values, using the intuitions of Section 3.1.1. We formalize these sets precisely in Section 3.4.

Finally, a constraint Δ is a conjunction of either type constraints Q or term equality constraints $x \approx e$, and in addition *strictness* constraints $x \approx \bot$. Strictness constraints are important for computing divergent sets for which we have used informal notation in the previous sections: For example, the value set abstraction $\{(\_ : \_) \ \bot\}$ is formally represented as $\{\Gamma \vdash (x : y) \ z \rhd z \approx \bot\}$, for some appropriate environment Γ.

Type constraints include type equalities $\tau_1 \sim \tau_2$ but can also include other constraints introduced by pattern matching or type signatures (examples would be type class constraints or refinements (Rondon et al., 2008; Vazou et al., 2014)). We leave the syntax of Q deliberately open.

3.3.2 Clause Processing

Our algorithm performs an abstract interpretation of the concrete dynamic semantics of Haskell pattern matching, and manipulates value vector abstractions instead of concrete value vectors. It follows the scheme described in Section 3.1 and illustrated in Figure 3.1. The key question is how *patVecProc* works; that is the subject of this section, and constitutes the heart of the chapter.

Initialization

As shown in Figure 3.1, the algorithm is initialized with a set U_0 representing "all values". For every function definition of the form:

$$\begin{array}{l} f :: \forall \vec{a}.\ \tau_1 \to \ldots \to \tau_n \to \tau \\ f\ p_{11}\ \ldots\ p_{1n} = \ldots \\ \qquad \ldots \\ f\ p_{m1}\ \ldots\ p_{mn} = \ldots \end{array}$$

the initial call to *patVecProc* will be with a singleton set:

$$U_0 = \{\vec{a}, (x_1 : \tau_1), \ldots, (x_n : \tau_n) \vdash x_1\ \ldots\ x_n \rhd \epsilon\}$$

As a concrete example, the pattern match clauses of function *zip* of type $\forall a.\ \forall b.\ [a] \to [b] \to [(a, b)]$ from Section 2.2.1 will be initialized with

$$U_0 = \{a, b, x_1 : [a], x_2 : [b] \vdash x_1\ x_2 \rhd \epsilon\}$$

Notice that we use variables x_i, rather than the underscores used informally in earlier sections, so that we can record their types in Γ, and constraints on their values in Δ.

Figure 3.5 Clause Processing

$$\boxed{\mathit{patVecProc}(\vec{p}, S) = \langle C, U, D \rangle}$$

$$\mathit{patVecProc}\ (\vec{p}, S) = \langle C, U, D \rangle \quad \text{where} \quad \begin{array}{l} C = \{ w \mid v \in S, w \in \mathcal{C}\ \vec{p}\ v,\ \vdash_{\mathrm{SAT}} w \} \\ U = \{ w \mid v \in S, w \in \mathcal{U}\ \vec{p}\ v,\ \vdash_{\mathrm{SAT}} w \} \\ D = \{ w \mid v \in S, w \in \mathcal{D}\ \vec{p}\ v,\ \vdash_{\mathrm{SAT}} w \} \end{array}$$

The Main Algorithm

Figure 3.5 gives the details of *patVecProc*. Given a pattern vector $\vec{p}$ and an incoming set S of value vector abstractions, *patVecProc* computes the sets C, U, D of covered, uncovered, and diverging values respectively. As Figure 3.5 shows, each is computed independently, in two steps. For each value vector abstraction v in S:

- *Use syntactic structure:* an auxiliary function ($\mathcal{C}$,$\mathcal{U}$ and $\mathcal{D}$) identifies the subset of v that is covered, uncovered, and divergent, respectively.
- *Use type and term constraints:* filter the returned set, retaining only those members whose constraints Δ are satisfiable.

We describe each step in more detail, beginning with the syntactic function for covered sets, C.

Computing the Covered Set

The function $\mathcal{C}\ \vec{p}\ v$ refines v into those vectors that are covered by the pattern vector $\vec{p}$. It is defined inductively over the structure of $\vec{p}$. Depending on the inputs, there are six cases to consider, each described below.

Rule CNil The first rule handles the case when both the pattern vector and the value vector are empty. In this case the value vector is trivially covered:

$$\mathcal{C}\ \epsilon\ (\Gamma \vdash \epsilon \triangleright \Delta) = \{\ \Gamma \vdash \epsilon \triangleright \Delta\ \}$$

Rule CConCon Next, Rule CConCon handles cases where both the pattern and value vector start with constructors K_i and K_j respectively. If the constructors differ ($K_i \neq K_j$), then this particular value vector *is not* covered so the covered set is empty:

$$\mathcal{C}\ ((K_i\ \vec{p})\ \vec{q})\ (\Gamma \vdash (K_j\ \vec{u})\ \vec{w} \triangleright \Delta) = \varnothing \qquad\qquad (K_i \neq K_j)$$

If the constructors are the same ($K_i = K_j$), then we proceed recursively with the subterms $\vec{p}$ and $\vec{u}$ and the suffixes $\vec{q}$ and $\vec{w}$:

$$\mathcal{C}\ ((K_i\ \vec{p})\ \vec{q})\ (\Gamma \vdash (K_i\ \vec{u})\ \vec{w} \triangleright \Delta) = \mathit{map}\ (\mathit{kcon}\ K_i)\ (\mathcal{C}\ (\vec{p}\ \vec{q})\ (\Gamma \vdash \vec{u}\ \vec{w} \triangleright \Delta))$$

Since we use a single recursive call where the pattern vectors are "flattened", we recover the original structure afterwards with auxiliary function $\mathit{kcon}\ K_i$, defined thus:

$$\mathit{kcon}\ K\ (\Gamma \vdash \vec{u}\ \vec{w} \triangleright \Delta) = \Gamma \vdash (K\ \vec{u})\ \vec{w} \triangleright \Delta$$

where $\vec{u}$ matches the arity of K.

Rule CConVar The third rule handles the case when the pattern vector starts with constructor K_i and the value vector with variable x:

$$\begin{array}{l} \mathcal{C}\ ((K_i\ \vec{p})\ \vec{q})\ (\Gamma \vdash x\ \vec{u} \triangleright \Delta) = \mathcal{C}\ ((K_i\ \vec{p})\ \vec{q})\ (\Gamma' \vdash (K_i\ \vec{y})\ \vec{u} \triangleright \Delta') \\ \quad \text{where} \quad \vec{y}\#\Gamma \quad \vec{a}\#\Gamma \quad (x : \tau_x) \in \Gamma \quad K_i :: \forall \vec{a}.\ Q \Rightarrow \vec{\tau} \to \tau \\ \qquad\qquad \Gamma' = \Gamma, \vec{a}, \vec{y} : \vec{\tau} \quad \Delta' = \Delta \cup Q \cup \tau \sim \tau_x \cup x \approx K_i\ \vec{y} \end{array}$$

In this case we refine x to the most general abstraction that matches the constructor, $K_i\ \vec{y}$, where the variables $\vec{y}$ are fresh for Γ, written $\vec{y}\#\Gamma$. Once the constructor shape for x has been exposed, Rule CConCon will fire to recurse into the pattern and value vectors. The constraint (Δ') used in the recursive call consists of the union of the original Δ with:

- Q; this is the constraint bound by the constructor $K_i :: \forall \vec{a}.\ Q \Rightarrow \vec{\tau} \to \tau$, which may for example include type equalities (in the case of GADTs).
- $x \approx K_i\ \vec{y}$; this records a term-level equality in the constraint that could be used by guard expressions.
- $\tau \sim \tau_x$, where τ_x is the type of x in the environment Γ, and τ is the return type of the constructor. This constraint will be useful when dealing with GADTs, as we explain in Section 3.3.3.

Rule CVar The fourth rule applies when the pattern vector starts with a variable pattern x. This matches any value abstraction u, so we can proceed inductively in $\vec{p}$ and $\vec{u}$:

$$\mathcal{C}\ (x\ \vec{p})\ (\Gamma \vdash u\ \vec{u} \rhd \Delta) = \mathit{map}\ (\mathit{ucon}\ u)\ (\mathcal{C}\ \vec{p}\ (\Gamma, x : \tau \vdash \vec{u} \rhd \Delta \cup x \approx u))$$

$$\text{where}\quad x \# \Gamma \quad \Gamma \vdash_{\scriptscriptstyle E} u : \tau$$

However, x may appear in some *guard* in the rest of the pattern, for example:

$$f\ x\ y \mid \mathit{Nothing} \leftarrow \mathit{lookup}\ x\ \mathit{env} = \ldots$$

To expose the fact that x is bound to u in subsequent guards (and in the right-hand side of the clause, see Section 3.3.6), Rule CVar adds $x \approx u$ to the constraints Δ, and correspondingly extends Γ to maintain the invariant that Γ binds all variables free in Δ. Finally, auxiliary function $\mathit{map}\ (\mathit{ucon}\ u)$ prefixes each of the recursive results with u:

$$\mathit{ucon}\ u\ (\Gamma \vdash \vec{u} \rhd \Delta) = \Gamma \vdash u\ \vec{u} \rhd \Delta$$

Rule CGuard The last case deals with the case where the first pattern in the pattern vector is a guard $(p \leftarrow e)$:

$$\mathcal{C}\ ((p \leftarrow e)\ \vec{p})\ (\Gamma \vdash \vec{u} \rhd \Delta) = \mathit{map}\ \mathit{tail}\ (\mathcal{C}\ (p\ \vec{p})\ (\Gamma, y : \tau \vdash y\ \vec{u} \rhd \Delta \cup y \approx e))$$

$$\text{where}\quad y \# \Gamma \quad \Gamma \vdash_{\scriptscriptstyle E} e : \tau$$

We make a recursive call to $\mathcal{C}$ adding p to the front of the pattern vector, and a fresh variable y to the front of the value abstraction. This variable y has the same type τ as e, and we add a term-equality constraint $y \approx e$ to the constraint set to record that y is bound to e. Finally, *map tail* removes the guard value from the returned value vector:

$$\mathit{tail}\ (\Gamma \vdash u\ \vec{u} \rhd \Delta) = \Gamma \vdash \vec{u} \rhd \Delta$$

Finally it is worth noting that the $\mathcal{C}\ \vec{p}\ v$ function always returns an empty or singleton set, but we use the full set notation for uniformity with the other functions.

Next, we discuss the other two functions ($\mathcal{U}$ and $\mathcal{D}$), which have a similar structure.

Computing the Uncovered Set

The function $\mathcal{U}\ \vec{p}\ v$ returns those vectors that are *not covered* by the pattern vector $\vec{p}$. Similarly to function $\mathcal{C}$, function $\mathcal{U}$ consists of six clauses, which we describe below.

Rule UNil When both the pattern vector and value vector are empty then (as we have seen in Rule CNil) the value vector is covered. Hence, the uncovered set is empty:

$$\mathcal{U}\ \epsilon\ (\Gamma \vdash \epsilon \rhd \Delta) = \varnothing$$

Rule UConCon Cases where both the pattern and value vector start with data constructors K_i and K_j are handled by Rule CConCon. If the head constructors match ($K_i = K_j$), we simply recurse:

$$\mathcal{U}\ ((K_i\ \vec{p})\ \vec{q})\ (\Gamma \vdash (K_i\ \vec{u})\ \vec{w} \rhd \Delta) = map\ (kcon\ K_i)\ (\mathcal{U}\ (\vec{p}\ \vec{q})\ (\Gamma \vdash \vec{u}\ \vec{w} \rhd \Delta)$$

If not ($K_i \neq K_j$), the value vector abstraction is uncovered, so we return it:

$$\mathcal{U}\ ((K_i\ \vec{p})\ \vec{q})\ (\Gamma \vdash (K_j\ \vec{u})\ \vec{w} \rhd \Delta) = \{\ \Gamma \vdash (K_j\ \vec{u})\ \vec{w} \rhd \Delta\ \} \qquad (K_i \neq K_j)$$

Rule UConVar Cases where the pattern vector starts with constructor K_i and the value vector with variable x are handled by Rule UConVar:

$$\begin{aligned}\mathcal{U}\ ((K_i\ \vec{p})\ \vec{q})\ (\Gamma \vdash x\ \vec{u} \rhd \Delta) &= \textstyle\bigcup_{K_j} \mathcal{U}\ ((K_i\ \vec{p})\ \vec{q})\ (\Gamma' \vdash (K_j\ \vec{y})\ \vec{u} \rhd \Delta') \\ \text{where}\quad & \vec{y}\#\Gamma \quad \vec{a}\#\Gamma \quad (x : \tau_x) \in \Gamma \quad K_j :: \forall \vec{a}.\ Q \Rightarrow \vec{\tau} \rightarrow \tau \\ & \Gamma' = \Gamma, \vec{a}, \vec{y} : \vec{\tau} \quad \Delta' = \Delta \cup Q \cup \tau \sim \tau_x \cup x \approx K_j\ \vec{y}\end{aligned}$$

Rule UConVar takes the union of the uncovered sets for all refinements of the variable x to a constructor K_j; each can lead recursively through Rule UConCon to uncovered cases. To inform guards, we record the equality $x \approx K_j\ \vec{y}$ for each constructor. As in Rule CConVar, we also record a type constraint between the constructor return type and the type of x in Γ.

Notice that Rule CConVar could handle this case similarly and recurse using all refinements of x. Yet, all refinements but one (the K_i) would fail matching using Rule CConCon in the next step. Thus, we opted for a simpler and more performant generation of the covered set.

Rule UVar Cases where the pattern vector starts with a variable are handled by Rule UVar. Since variable x can match against any possible term, the clause has identical structure to Rule CVar (modulo the recursive call to $\mathcal{U}$):

$$\begin{aligned}&\mathcal{U}\ (x\ \vec{p})\ (\Gamma \vdash u\ \vec{u} \rhd \Delta) = map\ (ucon\ u)\ (\mathcal{U}\ \vec{p}\ (\Gamma, x : \tau \vdash \vec{u} \rhd \Delta \cup x \approx u)) \\ &\quad \text{where}\quad x\#\Gamma \quad \Gamma \vdash_{\scriptscriptstyle E} u : \tau\end{aligned}$$

Rule UGuard Last, Rule UGUARD deals with guards, and has identical structure to the corresponding clause of $\mathcal{C}$ (Rule CGUARD):

$$\mathcal{U}\ ((p \leftarrow e)\ \vec{p})\ (\Gamma \vdash \vec{u} \rhd \Delta) = \mathit{map\ tail}\ (\mathcal{U}\ (p\ \vec{p})\ (\Gamma, y : \tau \vdash y\ \vec{u} \rhd \Delta \cup y \approx e))$$
$$\text{where}\quad y\#\Gamma \quad \Gamma \vdash_{\text{E}} e : \tau$$

Computing the Divergent Set

The function $\mathcal{D}\ \vec{p}\ v$ returns those vectors that diverge when matching the pattern vector $\vec{p}$. Following the structure of functions $\mathcal{C}$ and $\mathcal{U}$, function $\mathcal{D}$ has six clauses.

Rule DNil Firstly, the empty value vector does not diverge:

$$\mathcal{D}\ \epsilon\ (\Gamma \vdash \epsilon \rhd \Delta) = \varnothing$$

Rule DConCon In the case of constructors in the head of the pattern vector as well as the value vector (Rule DCONCON) there is no divergence either: we either recurse when the constructors match

$$\mathcal{D}\ ((K_i\ \vec{p})\ \vec{q})\ (\Gamma \vdash (K_i\ \vec{u})\ \vec{w} \rhd \Delta) = \mathit{map}\ (\mathit{kcon}\ K_i)\ (\mathcal{D}\ (\vec{p}\ \vec{q})\ (\Gamma \vdash \vec{u}\ \vec{w} \rhd \Delta)$$

or else return the empty divergent set:

$$\mathcal{D}\ ((K_i\ \vec{p})\ \vec{q})\ (\Gamma \vdash (K_j\ \vec{u})\ \vec{w} \rhd \Delta) = \varnothing \qquad\qquad (K_i \neq K_j)$$

Rule DConVar When the clause starts with constructor K_i, and the vector with a variable x, Rule DCONVAR combines two different results: (a) the first result represents symbolically all vectors where x diverges; (b) the second result recurses by refining x to $K_i\ \vec{y}$.

$$\begin{array}{rcl}\mathcal{D}\ ((K_i\ \vec{p})\ \vec{q})\ (\Gamma \vdash x\ \vec{u} \rhd \Delta) & = & \{\ \Gamma \vdash x\ \vec{u} \rhd \Delta \cup (x \approx \bot)\} \\ & \cup & \mathcal{D}\ ((K_i\ \vec{p})\ \vec{q})\ (\Gamma' \vdash (K_i\ \vec{y})\ \vec{u} \rhd \Delta')\end{array}$$
$$\begin{array}{rl}\text{where} & \vec{y}\#\Gamma \quad \vec{a}\#\Gamma \quad (x : \tau_x) \in \Gamma \quad K_i :: \forall \vec{a}.\ Q \Rightarrow \vec{\tau} \rightarrow \tau \\ & \Gamma' = \Gamma, \vec{a}, \vec{y} : \vec{\tau} \quad \Delta' = \Delta \cup Q \cup \tau \sim \tau_x \cup x \approx K_i\ \vec{y}\end{array}$$

In the first case we record the divergence of x with a *strictness* constraint $(x \approx \bot)$. For the second case, we appeal recursively to the divergent set computation (We give more details on the Δ' that we use to recurse in Sections 3.3.3 and 3.3.4).

In fact, this is the only clause of $\mathcal{D}$ which extends the divergent set: a constructor pattern is more refined than a variable (according to Definition 1, $p \not\preceq x$), and thus forces evaluation.

Rule DVar The case for variables (Rule DVar) is similar to the corresponding rules for $\mathcal{C}$ and $\mathcal{U}$:

$$\mathcal{D}\ (x\ \vec{p})\ (\Gamma \vdash u\ \vec{u} \rhd \Delta) = map\ (ucon\ u)\ (\mathcal{D}\ \vec{p}\ (\Gamma, x : \tau \vdash \vec{u} \rhd \Delta \cup x \approx u))$$
$$\text{where}\ \ x\#\Gamma \quad \Gamma \vdash_{\text{E}} u : \tau$$

Rule DGuard Finally, Rule DGuard deals with guards and is again similar to CGuard and UGuard:

$$\mathcal{D}\ ((p \leftarrow e)\ \vec{p})\ (\Gamma \vdash \vec{u} \rhd \Delta) = map\ tail\ (\mathcal{D}\ (p\ \vec{p})\ (\Gamma, y : \tau \vdash y\ \vec{u} \rhd \Delta \cup y \approx e))$$
$$\text{where}\ \ y\#\Gamma \quad \Gamma \vdash_{\text{E}} e : \tau$$

Filtering the Results with Constraints

Function *patVecProc* prunes the results of $\mathcal{C}\ \vec{p}\ v$, $\mathcal{U}\ \vec{p}\ v$, and $\mathcal{D}\ \vec{p}\ v$ that are semantically empty by appealing to an oracle judgment $\vdash_{\text{SAT}} (\Gamma \vdash \vec{u} \rhd \Delta)$. In the next section we define "semantically empty" by giving a denotational semantics to a value vector abstraction $[\![v]\!]$ as a set of concrete value vectors.

The purpose of $\vdash_{\text{SAT}}$ is to determine whether this set is empty. However, because satisfiability is undecidable in general (particularly when constraints involve term equivalence), we have to be content with a decidable algorithm $\vdash_{\text{SAT}}$ that gives sound but incomplete approximation to satisfiability:

$$\not\vdash_{\text{SAT}} v \quad \Rightarrow \quad [\![v]\!] = \emptyset$$

In terms of the outcomes (1-3) in Section 3.1.1, "soundness" means

1. If we do not warn that a set of clauses may be non-exhaustive, then they are definitely exhaustive.
2. If we warn that a clause is redundant, then it definitely is redundant.
3. If we warn that a right-hand side of a non-redundant clause is inaccessible, then it definitely is inaccessible.

Since $\vdash_{\text{SAT}}$ is necessarily incomplete, the converse does not hold in general. There is, of course, a large design space of less-than-complete implementations for $\vdash_{\text{SAT}}$. Our implementation is explained in detail in Chapter 4.

Another helpful insight is this: during constraint generation (in functions $\mathcal{C}$, $\mathcal{U}$, and $\mathcal{D}$) the sole purpose of adding constraints to Δ is to increase the chance that $\vdash_{\text{SAT}}$ will report "unsatisfiable". It is always sound to omit constraints from Δ so an implementation is free to trade off accuracy against the size of the constraint set.

3.3.3 Type Constraints from GADTs

Rules CConVar, UConVar, and DConVar record *type equalities* of the form $\tau \sim \tau_x$ between the value abstraction type (τ_x) and the return type of the appropriate data constructor each time (τ).

Recording these constraints in Rules CConVar and UConVar is important for reporting precise warnings when dealing with GADTs, as the following example demonstrates:

$$\begin{array}{ll}
\textbf{data}\ T\ a\ \textbf{where} & \mathit{foo} :: T\ c \to T\ c \to \mathit{Int} \\
\quad \mathit{TList}\ :: T\ [a] & \mathit{foo}\ \ \mathit{TList}\ \_\qquad = \ldots \\
\quad \mathit{TBool} :: T\ \mathit{Bool} & \mathit{foo}\ \ \_\qquad \mathit{TList}\ = \ldots
\end{array}$$

To determine C_2, the covered set from the second equation, we start from an initial singleton vector abstraction $U_0 = \{\Gamma_0 \vdash x_1\ x_2 \triangleright \epsilon\}$ with $\Gamma = c, x_1 : T\ c, x_2 : T\ c$. Next compute the uncovered set from the first clause, which (via UConVar and UVar) is $U_1 = \{\Gamma_1 \vdash \mathit{TBool}\ x_2 \triangleright \Delta_1\}$, where

$$\begin{array}{lcl}
\Gamma_1 & = & \Gamma_0, a \\
\Delta_1 & = & (x_1 \approx \mathit{TBool}) \cup (T\ c \sim T\ \mathit{Bool})
\end{array}$$

Note the recorded type constraint for the uncovered constructor *TBool* from Rule UConVar. Next, from U_1, compute the covered set for the second equation (via Rules CVar and CConVar):

$$\begin{array}{lcl}
C_2 & = & \mathcal{C}\ (\_\ \mathit{TList})\ (\Gamma_1 \vdash \mathit{TBool}\ x_2 \triangleright \Delta_1) \\
 & = & \{\Gamma_1, b \vdash \mathit{TBool}\ \ \mathit{TList} \triangleright \Delta_1 \cup (x_2 \approx \mathit{TList}) \cup (T\ c \sim T\ [b])\}
\end{array}$$

Note the type constraint ($T\ c \sim T\ [b]$) generated by rule CConVar. It allows us to rewrite constraint ($T\ c \sim T\ \mathit{Bool}$) to ($T\ [b] \sim T\ \mathit{Bool}$). The final constraint is unsatisfiable, C_2 is semantically empty, and the second equation is unreachable. Unless Rule CConVar or UConVar both record the type constraints we would miss reporting the second branch as redundant.

Rule DConVar also records term and type-level constraints in the recursive call. Indeed, if the first case in that rule is deemed unsatisfiable by our oracle it is important to have a precise set of constraints for the recursive call to detect possible semantic emptiness of the result.

3.3.4 Term Constraints from Guards

A major feature of our approach is that it scales nicely to handle *guards*, and other syntactic extensions of pattern-matching supported by GHC. Since to our knowledge there exists no other checking algorithm which handles guards, it might be illuminating to see how the rules work in practice. Consider again function abs_1 from Section 2.4.2. We may compute (laboriously) as follows:

$$\begin{aligned}
U_0 &= \{v : Int \vdash v \triangleright \epsilon\} \\
U_1 &= \mathcal{U}\ (x\ (True \leftarrow x < 0))\ (v : Int \vdash v \triangleright \epsilon) \\
&= \text{(apply UVAR)} \\
&\quad map\ (ucon\ v)\ (\mathcal{U}\ (True \leftarrow v < 0)\ (v : Int \vdash \epsilon \triangleright x \approx v)) \\
&= \text{(apply UGUARD)} \\
&\quad map\ (ucon\ v)\ (map\ tail \\
&\quad\quad (\mathcal{U}\ (True)\ (v : Int, y : Bool \vdash y \triangleright x \approx v, y \approx v < 0)) \\
&= \text{(apply UCONVAR; the } True/True \text{ case yields } \varnothing) \\
&\quad map\ (ucon\ v)\ (map\ tail\ (map\ (ucon\ y) \\
&\quad\quad (\mathcal{U}\ True\ (v : Int, y : Bool \vdash False \triangleright x \approx v, y \approx v < 0, y \approx False)) \\
&= \text{(apply UCONCON with } K_i \neq K_j\text{, and do the } maps) \\
&\quad \{v : Int, y : Bool \vdash v \triangleright x \approx v, y \approx v < 0, y \approx False\}
\end{aligned}$$

This correctly characterizes the uncovered values as those $v : Int$ for which $v < 0$ is *False*.

3.3.5 Extension 1: Smarter Initialization

Until now, we always initialized U_0 with the empty constraint, $\Delta = \epsilon$. But, consider these definitions:

```
type family F a                data T a where
type instance F Int = Bool       TInt  :: T Int
                                 TBool :: T Bool
```

Datatype T is a familiar GADT definition. F is a *type family*, or type-level function (Chakravarty et al., 2005a; Schrijvers et al., 2008),[5] equipped with an instance that declares $F\ Int = Bool$. Given these definitions, is the second clause of f below redundant?

```
f :: F a ~ b => T a -> T b -> Int
f TInt  TBool = ...
f TInt  x     = ...
f TBool y     = ...
```

Function f matches the first argument with *TInt*, yielding the local type equality $a \sim Int$. Using this fact, together with the signature constraint $F\ a \sim b$ and the top-level equation $F\ Int = Bool$, we can deduce that $Bool \sim b$, and hence the

[5] We elaborate more on type families in the second part of this thesis (Part II).

second clause is in fact redundant. In this reasoning we had to use the quantified constraint $F\ a \sim b$ from the signature of f. Hence the initial value abstraction U_0 for this pattern match should include constraints from the function signature:

$$U_0 = \{a, b, (x_1 : T\ a), (x_2 : T\ b) \vdash x_1\ x_2 \triangleright \mathbf{F\ a \sim b}\}$$

3.3.6 Extension 2: Nested Pattern Matches

Consider this definition:

$$\begin{array}{lll} f\ \ [] & = & \ldots \\ f\ \ x & = & \ldots (\mathbf{case}\ x\ \mathbf{of}\ \{\ w : ws \to e\ \}) \ldots \end{array}$$

The clauses of f and those of the inner case expression are disconnected. And yet we can see that the inner case expression is exhaustive, because the $x = []$ case is handled by the first equation.

Happily there is a principled way to allow the inner case to take advantage of knowledge from the outer one: *gather the constraints from the covered set of the outer pattern match, propagate them inwards, and use them to initialize* U_0 *for the inner one.* In this example, we may follow the algorithm as follows:

$$\begin{array}{l} U_0^f = \{a, v : [a] \vdash v \triangleright \varnothing\} \\ U_1^f = \{a, v : [a], v_1 : a, v_2 : [a] \vdash (v_1 : v_2) \triangleright \varnothing\} \\ C_2^f = \{a, v : [a], v_1 : a, v_2 : [a], x : [a] \vdash (v_1 : v_2) \triangleright x \approx v_1 : v_2\} \end{array}$$

Propagate C_2^f inwards to the case expression. Now initialize the U_0^{case} for the case expression thus:

$$U_0^{case} = \{(\Gamma \vdash x \triangleright \Delta) \mid (\Gamma \vdash \vec{u} \triangleright \Delta) \in C_2^f\}$$

It can be seen that the Δ used for the inner case will include the constraint $x = v_1 : v_2$ inherited from C_2^f, and that in turn can be used by $\vdash_{\mathrm{SAT}}$ to show that the $[]$ missing branch of the case is inaccessible. Notice that U_0 may now have more than one element; until now it has always been a singleton.

The same idea works for type equalities, so that type-equality knowledge gained in an outer pattern-match can be carried inwards in Δ and used to inform inner pattern matches.

Figure 3.6 Values and Value Typing

$$\begin{array}{lcll} \tau_c & ::= & T\ \overline{\tau}_c \mid \tau_c \to \tau_c & \textit{closed monotype} \\ V, W & ::= & K\ \vec{V} \mid \lambda x.\ e \mid \bot & \textit{value} \end{array}$$

$\boxed{\vdash_{\mathrm{V}} V : \tau_c}$ Well-typed Values

$$\frac{}{\vdash_{\mathrm{V}} \bot : \tau_c}\ \textsc{Bot} \qquad \frac{x : \tau_{c_1} \vdash_{\mathrm{E}} e : \tau_{c_2}}{\vdash_{\mathrm{V}} \lambda x.\ e : \tau_{c_1} \to \tau_{c_2}}\ \textsc{Fun}$$

$$\frac{\begin{array}{c} K :: \forall \overline{a}.\ \forall \overline{b}.\ Q \Rightarrow \overline{\tau} \to T\ \overline{a} \\ \models \theta(Q) \qquad \theta = [\overline{\tau}_{c_i}/\overline{a}, \overline{\tau}_{c_j}/\overline{b}] \qquad \vdash_{\mathrm{V}} V_i : \theta(\tau_i) \quad (\forall i) \end{array}}{\vdash_{\mathrm{V}} K\ \vec{V} : T\ \vec{\tau}_{c_i}}\ \textsc{Con}$$

3.4 Meta-theory

In order to formally relate the algorithm to the dynamic semantics of pattern matching, we first formalize the latter as well as the semantics of the value abstractions used by the former.

3.4.1 Values and Typing

As outlined in Section 3.3.1 a value abstraction denotes a set of values. Hence, before we give the denotation of value abstractions, we briefly discuss the syntax and typing of values, which is given in Figure 3.6. The syntax of values V comprises constructor values ($K\ \vec{V}$), λ-expressions ($\lambda x.\ e$) and—since we are targeting call-by-name semantics—bottom values ($\bot$). Relation $\vdash_{\mathrm{V}} V : \tau_c$ assigns to a value V a closed monotype τ_c and is entirely standard.[6]

3.4.2 Value Abstractions

The denotation of value abstractions as a set of values is formalized in Figure 3.7.

As the figure shows, the meaning of a closed value abstraction $\Gamma \vdash \vec{u} \rhd \Delta$ is the set of all type-respecting instantiations of $\vec{u}$ to a vector of (closed) values

[6]Rule Fun also makes an appeal to the term typing relation $\Gamma \vdash_{\mathrm{E}} e : \tau$. The definition of the term typing relation is out of the scope of this chapter and hence omitted.

Figure 3.7 Semantics of Value Abstractions and Patterns

$$\mathscr{S}, \mathscr{C}, \mathscr{U}, \mathscr{D} \ ::= \ \overline{\vec{V}} \qquad \textit{set of value vectors}$$

$\boxed{[\![S]\!] = \overline{\vec{V}}}$ Denotation of Value Abstractions

$$[\![S]\!] = \{\theta(\vec{u}) \mid (\Gamma \vdash \vec{u} \rhd \Delta) \in S,\ \theta \in [\![\Gamma]\!],\ \models \theta(\Delta)\}$$

$\boxed{[\![\Gamma]\!] = \overline{\theta}}$ Denotation of Typing Environments

$$\begin{array}{lcl} [\![\epsilon]\!] & = & \{\epsilon\} \\ [\![x : \tau_c, \Gamma]\!] & = & \{\theta \cdot [V/x] \ \mid \vdash_{\mathrm{V}} V : \tau_c, \theta \in [\![\Gamma]\!]\} \\ [\![a, \Gamma]\!] & = & \{\theta \cdot [\tau_c/a] \ \mid \ \theta \in [\![[\tau_c/a](\Gamma)]\!]\} \end{array}$$

$\boxed{[\![\vec{p}]\!]^\theta :: \vec{V} \to M}$ Denotation of Pattern Vectors (Auxiliary)

$$\begin{array}{ll} [\![\epsilon]\!]^\theta(\epsilon) & = \mathcal{T} \\ [\![x\ \vec{p}]\!]^\theta(V\ \vec{V}) & = [\![\vec{p}]\!]^{[V/x]\cdot\theta}(\vec{V}) \\ [\![(p \leftarrow e)\ \vec{p}]\!]^\theta(\vec{V}) & = [\![p\ \vec{p}]\!]^\theta(E[\![\theta(e)]\!]\ \vec{V}) \\ [\![(K_i\ \vec{p})\ \vec{q}]\!]^\theta((K_j\ \vec{V})\ \vec{W}) & = \begin{cases} [\![\vec{p}\ \vec{q}]\!]^\theta(\vec{V}\ \vec{W}) & \text{, if } K_i = K_j \\ \mathcal{F} & \text{, if } K_i \neq K_j \end{cases} \\ [\![(K_i\ \vec{p})\ \vec{q}]\!]^\theta(\bot\ \vec{V}) & = \bot \end{array}$$

$\boxed{[\![\vec{p}]\!] :: \overline{\vec{V}} \to \langle \overline{\vec{V}}_c, \overline{\vec{V}}_u, \overline{\vec{V}}_\bot \rangle\rangle}$ Denotation of Pattern Vectors

$$\begin{array}{lcl} [\![\vec{p}]\!](\mathscr{S}) & = & \langle\{\vec{V} \mid \vec{V} \in \mathscr{S} \text{ where } [\![\vec{p}]\!]^\epsilon(\vec{V}) = \mathcal{T}\} \\ & & ,\{\vec{V} \mid \vec{V} \in \mathscr{S} \text{ where } [\![\vec{p}]\!]^\epsilon(\vec{V}) = \mathcal{F}\} \\ & & ,\{\vec{V} \mid \vec{V} \in \mathscr{S} \text{ where } [\![\vec{p}]\!]^\epsilon(\vec{V}) = \bot\}\rangle \end{array}$$

$\vec{V} = \theta(\vec{u})$, such that the constraints $\theta(\Delta)$ are satisfied. The judgment $\models \Delta$ denotes the logical entailment of the (closed) constraints Δ; we omit the details of its definition for the sake of brevity.

A "type-respecting instantiation", or denotation, of a type environment Γ is a substitution θ whose domain is that of Γ; it maps each type variable $a \in \Gamma$ to a closed type; and each term variable $x : \tau \in \Gamma$ to a closed value V of the appropriate type $\vdash_v V : \tau$. For example

$$\begin{array}{llllllllll}
\multicolumn{10}{l}{[\![\{\ a, b, x : a, y : b \vdash x\ y \rhd a \sim \mathit{Bool}, b \sim ()\ \}]\!]} \\
= \{ & \mathit{True} & () & , & \mathit{False} & () & , & \bot & () & , \\
 & \mathit{True} & \bot & , & \mathit{False} & \bot & , & \bot & \bot & \}
\end{array}$$

3.4.3 Pattern Vectors

Figure 3.7 also formalizes the dynamic semantics of pattern vectors.

The basic meaning $[\![\vec{p}]\!]^\theta$ of a pattern vector $\vec{p}$ is a function that takes a vector of values $\vec{V}$ to a matching result M. There may be free variables in (the guards of) $\vec{p}$; the given substitution θ binds them to values. The matching result M has the form $\mathcal{T}$, $\mathcal{F}$ or $\bot$ depending on whether the match succeeds, fails or diverges:[7]

$$M \ ::= \ \mathcal{T} \mid \mathcal{F} \mid \bot \qquad\qquad \textit{matching result}$$

Consider matching the pattern vector $x\ (\mathit{True} \leftarrow x > y)$, where y is bound to 5, against the value 7; this match succeeds. Formally, this is expressed thus:

$$[\![x\ (\mathit{True} \leftarrow x > y)]\!]^{[5/y]}(7) = \mathcal{T}$$

For comparing with our algorithm, this formulation of the dynamic semantics is not ideal: the former acts on whole sets of value vectors (in the form of value abstractions) at a time, while the latter considers only one value vector at a time. To bridge this gap, $[\![\vec{p}]\!]$ lifts $[\![\vec{p}]\!]^\epsilon$ from an individual value vector $\vec{V}$ to a whole set S of value vectors. It does so by partitioning the set based on the matching outcome, which is similar to the behavior of our algorithm.

3.4.4 Correctness Theorem

Now we are ready to express the correctness of the algorithm with respect to the dynamic semantics. The algorithm is essentially an abstract interpretation of the dynamic semantics. Rather than acting on an infinite set of values, it acts on a finite representation of that set, the value abstractions. Correctness amounts to showing that the algorithm treats the abstract set in a manner that faithfully reflects the way the dynamic semantics treats the corresponding concrete set. In other words, it should not matter whether we run the algorithm on an abstract set S and interpret the abstract result $\langle C, U, D \rangle$ as sets of

[7]We omit the denotation of expressions $E[\![e]\!] = V$, which depends on the selection of e.

concrete values $\langle \mathscr{C}, \mathscr{U}, \mathscr{D} \rangle$, or whether we first interpret the abstract set S as a set $\mathscr{S}$ of concrete values and then run the concrete dynamic semantics on those.

This can be expressed concisely as a commuting diagram:

$$\begin{tikzcd}
S \arrow[r, "patVecProc(\vec{p})"] \arrow[d, "\llbracket\cdot\rrbracket"'] & \langle C, U, D \rangle \arrow[d, "\llbracket\cdot\rrbracket"] \\
\mathscr{S} \arrow[r, "\llbracket\vec{p}\rrbracket"'] & \langle \mathscr{C}, \mathscr{U}, \mathscr{D} \rangle
\end{tikzcd}$$

This diagram allows us to interpret the results of the algorithm. For instance, if we choose s to cover all possible value vectors and we observe that C is empty, we can conclude that no value vector successfully matches $\vec{p}$.

To state correctness precisely we have to add the obvious formal fine print about types: the behavior of pattern matching is only defined if:

1. the pattern vector $\vec{p}$ is well-typed,
2. the value vector $\vec{V}$ and, by extension, the value set $\mathscr{S}$ and the abstract value set S are well-typed, and
3. the types of pattern vector $\vec{p}$ and value vector $\vec{V}$ correspond.

The first condition we express concisely with the judgment $Q; \Gamma \vdash \vec{p} : \vec{\tau}$, which expresses that the pattern vector $\vec{p}$ has types $\vec{\tau}$ for a type environment Γ and given type constraints Q.

For the second condition, we first consider the set of all values value vectors compatible with types $\vec{\tau}$, type environment Γ and given type constraints Q. This set can be compactly written as the interpretation $\llbracket S^* \rrbracket$ of the value abstraction $S^* = \{\Gamma, \vec{x} : \vec{\tau} \vdash \vec{x} \triangleright Q\}$. Any other well-typed value vectors $\vec{V}$ must be contained in this set: $\vec{V} \in \llbracket S^* \rrbracket$. Similarly, $\mathscr{S} \subseteq \llbracket S^* \rrbracket$ and $\llbracket S \rrbracket \subseteq \llbracket S^* \rrbracket$.

Finally, the third condition is implicitly satisfied by using the same types $\vec{\tau}$, type environment Γ and given type constraints Q.

Wrapping up, we formally state the correctness theorem as follows:

Theorem 1 (Correctness)**.**

$$\begin{array}{l}\forall \Gamma, Q, \vec{p}, \vec{\tau}, S : \\ \quad Q; \Gamma \vdash \vec{p} : \vec{\tau} \ \wedge \ [\![S]\!] \subseteq [\![\{\Gamma, \vec{x} : \vec{\tau} \vdash \vec{x} \triangleright Q\}]\!] \Longrightarrow [\![\mathit{patVecProc}(\vec{p}, S)]\!] = [\![\vec{p}]\!][\![S]\!]\end{array}$$

The proof of Theorem 1 is future work.

3.4.5 Complexity

Every pattern-checking algorithm has terrible worst-case complexity, and ours is no exception. For example, consider

$$\begin{array}{l}\textbf{data}\ T = A \mid B \mid C \\ f\ \ A\ \ A\ \ =\ \ \mathit{True} \\ f\ \ B\ \ B\ \ =\ \ \mathit{True} \\ f\ \ C\ \ C\ \ =\ \ \mathit{True}\end{array}$$

What values U_3 are not covered by f? Answer

$$\{A\ B, A\ C, B\ A, B\ C, C\ A, C\ B\}$$

The size of the uncovered set is the square of the number of constructors in T. It gets worse: Sekar et al. (1995) show that the problem of finding redundant clauses is NP-complete, by encoding the boolean satisfiability (SAT) problem into it. So the worst-case running time is necessarily exponential. But so is Hindley-Milner type inference! As with type inference, we hope that worst case behavior is rare in practice. Moreover, GHC's current redundancy checker suffers from the same problem without obvious problems in practice. We have gathered quantitative data about set sizes to better characterize the problem, which we discuss in the next chapter (Section 4.5.1).

3.5 Related Work

3.5.1 Compiling Pattern Matching

There is a large body of work concerned with the *efficient compilation* of pattern matching, for strict and lazy languages (Laville, 1991; Maranget, 1992; Maranget and Para, 1994; Maranget, 2008). Although superficially related, these works focus on an entirely different problem, one that simply does not arise for us.

Consider

$$\begin{array}{llll} f & \mathit{True} & \mathit{True} & = 1 \\ f & \_ & \mathit{False} & = 2 \\ f & \mathit{False} & \mathit{True} & = 3 \end{array}$$

In a strict language one can choose whether to begin by matching the first argument or the second; the choice affects only efficiency, not semantics. In a lazy language the choice affects semantics; for example, does $(f\ \bot,\ \mathit{False})$ diverge, or return 2? Laville and Maranget suggest choosing a match order that makes f maximally defined (Maranget, 1992), and they explore the ramifications of this choice.

However, Haskell does not offer this degree of freedom; it fixes a top-to-bottom and left-to-right order of evaluation in pattern match clauses.

3.5.2 Warnings for Simple Patterns

We now turn our attention to *generating warnings* for inexhaustive or redundant patterns. For simple patterns (no guards, no GADTs) there are several related works. The most closely-related is Maranget's elegant algorithm for detecting missing and redundant (or "useless") clauses (Maranget, 2007).[8] Maranget recursively defines a predicate that determines whether there could be any vector of values v that matches pattern vector $\vec{p}$, without matching any pattern vector row in a matrix P, $U_{req}(P\ \vec{p})$, and answers both questions of exhaustiveness (query $\mathcal{U}_{req}(P\ \_)$) and redundancy (query $\mathcal{U}_{req}(P^{1..(j-1)}\ \vec{p_j})$ where $P^{1..(j-1)}$ corresponds to all previous clauses). Our algorithm has many similarities (e.g. in the way it expands constructor patterns) but is more incremental by propagating state from one clause to the next instead of examining all previous clauses for each clause.

Maranget's algorithm does not deal with type constraints (as those arising from GADTs), nor guards and nested patterns that require keeping track of Δ and environment Γ. Finally the subtle case of an empty covered set but a non-empty divergent set would not be treated specially (and the clause would be considered as non-redundant, though it could only allow values causing divergence).

Krishnaswami (2009) accounts for exhaustiveness and redundancy checking as part of a formalization of pattern matching in terms of the focused sequent calculus. His approach assumes a left-to-right ordering in the translation of ML patterns, which is compatible with Haskell's semantics.

[8]See also the discussion of Section 2.2.2.

Sestoft (1996) focuses on compiling pattern matches for a simply-typed variant of ML, but his algorithm also identifies inexhaustive matches and redundant match rules as a by-product.

3.5.3 Warnings for GADT Patterns

OCaml and Idris both support GADTs, and both provide some GADT-aware support for pattern-match checking. No published work describes the algorithm used in these implementations.

OCaml When Garrigue and Normand (2011) introduced GADTs to the OCaml language, they also extended the checking algorithm. It eliminated the ill-typed uncovered cases proposed by OCaml's original algorithm. However, their approach did not identify clauses that were redundant due to unsatisfiable type constraints. For instance, the third clause in f below was not identified as redundant.

```
type _ t = T1 : int t | T2 : bool t

let f (type a) (x: a t) (y: a s) : unit =
  match (x, y) with
   | (T1, T1) -> ()
   | (T2, T2) -> ()
   | (_,  _)  -> ()
```

Furthermore, this approach did not perform a redundancy check, which led to a discrepancy between the issued missing/redundant clauses.

In recent work (Garrigue and Normand, 2015) they presented an improved algorithm which gives much better results in matches involving GADTs. The two main contributions of this work are (a) the observation that pattern match checking for strict languages is equivalent to type inhabitation checking (and thus undecidable), and (b) the development of a coverage checking algorithm which uses backtracking to turn type inference into a proof search. Indeed, as we discuss in Section 4.2, we can reuse these findings to perform coverage checking of strict matches in Haskell too.

Idris Idris (Brady, 2013) has very limited checking of overlapping patterns and redundant patterns.[9] It does, however, check coverage, and will use this information in optimization and code generation.

ML Variants Xi (2003, 1998a,b) shows how to eliminate dead code for GADT pattern matching—and dependent pattern matching in general—for Dependent ML. He has a two-step approach: first add all the missing patterns using simple-pattern techniques (Section 3.5.2), and then prune out redundant clauses by checking when typing constraints are un-satisfiable. We combine the two steps, but the satisfiability checking is similar.

Dunfield's thesis (Dunfield, 2007b, Chapter 4) presents a coverage checker for Stardust (Dunfield, 2007a), another ML variant with refinement and intersection types. The checker proceeds in a top-down, left-to-right fashion much like Figure 3.1 and uses type satisfiability to prune redundant cases.

Neither of these works handles guards or laziness.

3.5.4 Constraint-based Exhaustiveness Checking

The idea of constraint-solving-based exhaustiveness checking employed by our algorithm can also be found in other works, some of which we discuss below.

Dialyzer Sagonas et al. (2013) extend Dialyzer's[10] analysis that infers success typings to provide more precise error detection for Erlang, by means of program slicing. Their 3-phase technique (a) generates subtyping constraints from the source program, (b) employs a constraint solver to detect inconsistencies in the generated constraints, and (c) post-processes inconsistencies found in phase (b) to refine the source of the inconsistencies and accurately pinpoint the source of run-time errors. Though the shape of constraints we generate is different (equality vs. subtyping constraints) and Sagonas et al. deal with a dynamically-typed language (Erlang), both works can detect non-exhaustive matches in their respective settings, by means of a generate-and-solve approach.

Since Erlang has strict semantics and a dynamic type system, Dialyzer does not deal with GADTs or laziness but can reason about boolean guards found in Erlang.

[9] Edwin Brady, personal communication.

[10] Dialyzer is the most advanced static analysis tool developed for Erlang. For more details on Dialyzer's internals see the work of Lindahl and Sagonas (2004).

Exception Analysis for Call-by-name Languages Koot and Hage (2015) develop a modular constraint-based exception analysis which gives accurate warnings about run-time errors due to non-exhaustive matches in call-by-name languages. Their analysis employs both data-flow analysis (by means of subtyping constraints) and control-flow analysis (by means of conditional constraints). Furthermore, it uses parametric polyvariance to achieve context-sensitivity and polyvariant recursion to avoid poisoning.

Though very similar to the work of Sagonas et al. (2013), this work targets a call-by-name semantics, which makes it more closely related to the work of our own. Nevertheless, Koot and Hage (2015) do not handle GADTs, pattern guards, or any other guard-like feature as we do (they deal with `if` clauses though).

3.5.5 Total Languages

Total languages like Agda (Norell, 2007) and Coq (The Coq development team, 2004) must treat non-exhaustive pattern matches as an *error* (not a warning). Moreover, they also allow overlapping patterns and use a variation of Coquand's dependent pattern matching (Coquand, 1992) to report redundant clauses. The algorithm works by splitting the context, until the current neighborhood matches one of the original clauses. If the current neighborhood fails to match all the given clauses, the pattern match is non-exhaustive and a coverage failure error is issued. If matching is inconclusive though, the algorithm splits along one of the blocking variables and proceeds recursively with the resulting neighborhoods. Finally, the `with`-construct (Norell, 2007), first introduced by McBride and McKinna (2004), provides (pattern) guards in a form that is suitable for total languages.

The key differences between our work and work on dependent pattern matching are the following: (i) because of the possibility of divergence we have to take laziness into account; (ii) current presentations of `with`-clauses (McBride and McKinna, 2004) do not introduce term-level equality propositions and hence may report inexhaustiveness checking more often than necessary, (iii) our approach is easily amenable to external decision procedures that are proven sound but do not have to return proof witnesses in the proof theory at hand.

3.5.6 Verification Tools

ESC/Haskell A completely different but more powerful approach can be found in *ESC/Haskell* (Xu, 2006) and its successor (Xu et al., 2009). ESC/Haskell is based on preconditions and contracts, so, it is able to detect far more defects in programs: pattern matching failures, division by zero, out of bounds array indexing, etc. Although it is far more expressive than our approach (e.g., it can verify even some sorting algorithms), it requires additional work by the programmer through explicit pre/post-conditions.

Catch Another approach that is closer to our work but retains some of the expressiveness of ESC/Haskell is the *Catch* tool (Mitchell and Runciman, 2008) Catch generates pre- and post-conditions that describe the sets of incoming and returned values of functions (quite similarly to our value abstraction sets). Catch is based on abstract interpretation over Haskell terms—the scope of abstract interpretation in our case is restricted to clauses (and potentially nested patterns). A difference is that Catch operates at the level of Haskell Core, GHC's intermediate language (Yorgey et al., 2012). The greatest advantage of this approach is that this language has only 10 data constructors, and hence Catch does not have to handle the more verbose source Haskell AST. Unfortunately, at the level of Core, the original syntax is lost, leading to less comprehensive error messages. On top of that, Catch does not take into account type constraints, such as those that arise from GADT pattern matching. Our approach takes them into account and reuses the existing constraint solver infrastructure to discharge them.

Liquid Types Liquid types (Rondon et al., 2008; Vazou et al., 2014) is a refinement types extension to Haskell. Similarly to ESC/Haskell, it can be used to detect redundant, overlapping, or non-exhaustive patterns, using an SMT-based version of Coquand's algorithm (Coquand, 1992). To take type-level constraints (such as type equalities from GADTs) into account, one would have to encode them as refinement predicates. The algorithm that we propose for computing covered, uncovered, and diverging sets would still be applicable, but would have to emit constraints in the vocabulary of Liquid types.

3.5.7 Algorithm Extensions

After the publication of our algorithm (Karachalias et al., 2015), we identified two developments that have been inspired by our work.

Z3 SMT Solver As we discussed in Section 3.1.3, one of the main benefits of our approach is the modular solving of constraints: one can use different oracles without affecting the implementation of the main algorithm at all.

Pavel Kalvoda and Tom Sydney Kerckhove recently implemented a proof-of-concept implementation of our algorithm in the form of a tool, which uses the Z3 SMT solver (developed by Microsoft Research) as a term oracle, for accurate warnings in the presence of guards. The source-code of the tool can be found in the following link:

`https://github.com/PJK/haskell-pattern-matching`

LambdaCube 3D Furthermore, the compiler of LambdaCube 3D (a Haskell-like purely functional domain specific language for GPU programming) uses a pattern match compiler and checker inspired by the algorithm of Section 3.3.2.[11] The key idea is to represent complex pattern matching structures as *guard trees*, following a syntax similar to that of Section 3.2.1. We believe that such an extension offers opportunities for new pattern matching features, like *or-patterns*. Indeed, we have independently investigated a similar extension: *pattern trees*. We elaborate more on this topic in Section 10.3.1, which discusses ongoing and future work.

The aforementioned algorithm is already implemented in the LambdaCube 3D compiler, which can be found here:

`https://github.com/lambdacube3d/lambdacube-compiler`

and an example-driven presentation of the algorithm can be found here:[12]

`https://goo.gl/xV22mS`

3.6 Scientific Output

This chapter has presented (a) a small—yet highly expressive—core pattern language in which we can encode a multitude of source-level features from the literature, and (b) a type-aware pattern match checking algorithm which processes programs written in this language and emits accurate warnings for missing or redundant patterns.

The main novelty of our approach lies in the use of abstract interpretation to capture sets of values, and the separation of concerns: the algorithm is

[11] Péter Diviánszky, personal communication.

[12] As presented by Péter Diviánszky (one of LambdaCube 3D lead developers) at the Budapest Haskell User Group meetup on April 7, 2016.

parameterized over an oracle that can solve type and term constraints. This allows the algorithm to focus on the structural aspects of pattern matching, while the oracle deals with constraints that arise from guards or GADTs. Thus, accommodation of type system extensions or better precision of the reported warnings can be achieved without affecting the main algorithm at all. Finally, we formalize the correctness of our algorithm with respect to the Haskell semantics of pattern matching.

Most of the material found in this chapter is drawn from the following publication:

> Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis and Simon Peyton Jones (2015). GADTs Meet Their Match: Pattern-matching Warnings That Account for GADTs, Guards, and Laziness. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, pp. 424-436, Vancouver, BC, Canada, August 31-September 2, 2015.

Within this multi-author work, the contribution of each author has been the following: The first version of the pattern match checking algorithm dealt only with laziness and GADTs, and was developed by the author of this thesis. This version was later collaboratively extended by all authors with the core pattern language, so that it could accommodate guards and guard-like features. The implementation in GHC has been developed by the author of this thesis, with the support of other GHC developers (in particular Ben Gamari).

Chapter 4

GHC Implementation

"If debugging is the process of removing software bugs, then programming must be the process of putting them in."

—Edsger Dijkstra

In order to evaluate our pattern match checking algorithm, we have implemented it in the cutting-edge Haskell compiler, GHC. In this chapter we elaborate on the instantiation of the theoretical model of Chapter 3 so that it can use the existing GHC infrastructure. Furthermore, we evaluate our implementation's performance, precision, and impact on several Hackage libraries. Given that GHC is actively being developed by several community members from academia and industry, the results we present here reflect the implementation of the pattern match checker at an earlier time. More specifically, the material found in this chapter reflects the implementation of our initial prototype (as described in the corresponding publication (Karachalias et al., 2015)), and the subsequent development until February 2017. GHC has evolved significantly since our initial implementation so it is very challenging to evaluate our implementation in isolation of the current compiler state.

The remainder of this chapter discusses several aspects of the implementation of our algorithm in GHC and is structured as follows: Section 4.1 presents a more concise and efficient formalization of the algorithm we presented in Section 3.3, which constitutes the starting point of our implementation. Section 4.2 discusses how our implementation deals with an extension we have not considered in the design of our algorithm: empty case expressions. Section 4.3 elaborates on several optimizations we have implemented to achieve competitive performance. Section 4.4 discusses the instantiation of the oracle $\vdash_{\text{SAT}}$. Section 4.5 demonstrates the effectiveness and evaluates the performance of the new checker on a set of actual Haskell programs submitted by GHC users, for whom inaccurate warnings were troublesome. Finally, concluding Section 4.6 summarizes our results.

4.1 Alternative Formalization

We have optimized the presentation of our algorithm in Sections 3.2 and 3.3 for clarity, rather than runtime performance. Even though we cannot improve upon the asymptotic worst-case time complexity, various measures can significantly improve the average performance of the algorithm. Additionally, we made several simplifications for the sake of readability. Of course, our implementation needs to handle all special cases that have not been covered in Sections 3.2 and 3.3. To this end, before discussing the optimizations of the algorithm we have implemented in GHC, in this section we present an alternative formalization of the main algorithm (i.e., functions $\mathcal{C}$, $\mathcal{U}$, and $\mathcal{D}$) which is more suitable for a performant implementation. The optimizations are discussed at length in Section 4.3.

4.1.1 Clause Processing

First, we utilize the observation that functions $\mathcal{C}$, $\mathcal{U}$, and $\mathcal{D}$ of Section 3.3.2 have a very similar structure to merge them into a single function to avoid multiple traversals: $\textsc{Match}(\vec{p},\ v) \Downarrow \mathcal{R}$. By $\mathcal{R}$ we denote the result of pattern match checking: $\mathcal{R}$ is simply a triple containing the covered, uncovered, and divergent sets:

$$\mathcal{R} \ ::= \ \langle C, U, D \rangle \qquad\qquad \textit{pattern match checking result}$$

The definition of judgment $\textsc{Match}(\vec{p},\ v) \Downarrow \mathcal{R}$ is given in Figure 4.1. Each rule combines the three corresponding rules of functions $\mathcal{C}$, $\mathcal{U}$, and $\mathcal{D}$. Rules ConEq and ConNe taken together capture Rules CConCon, UConCon, and DConCon: Rule ConEq handles cases where the data constructors are equal, while Rules ConNe cases where they differ.

The definition of auxiliary function $map_{\mathcal{R}}$ is straightforward:

$$map_{\mathcal{R}}\ f\ \langle C, U, D \rangle = \langle map\ f\ \ C, map\ f\ \ U, map\ f\ \ D \rangle$$

The only noticeable difference between the definition of $\textsc{Match}(\vec{p},\ v) \Downarrow \mathcal{R}$ and functions $\mathcal{C}$, $\mathcal{U}$, and $\mathcal{D}$ is captured in Rule ConVar.

Figure 4.1 Pattern Match Checking (All-In-One)

$$\boxed{\textsc{Match}(\vec{p},\ v) \Downarrow \mathcal{R}}$$

$$\frac{}{\textsc{Match}(\epsilon,\ \Gamma \vdash \epsilon \rhd \Delta) \Downarrow \langle \{\Gamma \vdash \epsilon \rhd \Delta\}, \varnothing, \varnothing \rangle}\ \textsc{Nil}$$

$$\frac{\textsc{Match}(\vec{p}\ \vec{q},\ \Gamma \vdash \vec{u}\ \vec{w} \rhd \Delta) \Downarrow \mathcal{R}}{\textsc{Match}((K_i\ \vec{p})\ \vec{q},\ \Gamma \vdash (K_i\ \vec{u})\ \vec{w} \rhd \Delta) \Downarrow (\mathit{map}_{\mathcal{R}}\ (\mathit{kcon}\ K_i)\ \mathcal{R})}\ \textsc{ConEq}$$

$$\frac{K_i \neq K_j}{\textsc{Match}((K_i\ \vec{p})\ \vec{q},\ \Gamma \vdash (K_j\ \vec{u})\ \vec{w} \rhd \Delta) \Downarrow \langle \varnothing, \{\Gamma \vdash (K_j\ \vec{u})\ \vec{w} \rhd \Delta\}, \varnothing \rangle}\ \textsc{ConNe}$$

$$\frac{x \notin \mathit{dom}(\Gamma) \qquad \Gamma \vdash u : \tau \qquad \textsc{Match}(\vec{p},\ \Gamma, x : \tau \vdash \vec{u} \rhd \Delta \cup x \approx u) \Downarrow \mathcal{R}}{\textsc{Match}(x\ \vec{p},\ \Gamma \vdash u\ \vec{u} \rhd \Delta) \Downarrow (\mathit{map}_{\mathcal{R}}\ (\mathit{ucon}\ u)\ \mathcal{R})}\ \textsc{Var}$$

$$\frac{\begin{array}{c} y \notin \mathit{dom}(\Gamma) \\ \Gamma \vdash e : \tau \qquad \textsc{Match}(\vec{p}\ \vec{q},\ \Gamma, y : \tau \vdash y\ \vec{u} \rhd \Delta \cup y \approx e) \Downarrow \mathcal{R} \end{array}}{\textsc{Match}((\vec{p} \leftarrow e)\ \vec{q},\ \Gamma \vdash \vec{u} \rhd \Delta) \Downarrow (\mathit{map}_{\mathcal{R}}\ \mathit{tail}\ \mathcal{R})}\ \textsc{Guard}$$

$$\frac{\begin{array}{c} \vec{y} \notin \mathit{dom}(\Gamma) \qquad \vec{a} \notin \mathit{dom}(\Gamma) \qquad \Gamma \vdash x : \tau_x \qquad K_j :: \forall \vec{a}.\ Q \Rightarrow \vec{\tau} \rightarrow \tau \\ \Gamma' = \Gamma, \vec{a}, \vec{y} : \vec{\tau} \qquad \Delta' = \Delta \cup Q \cup (\tau \sim \tau_x) \cup (x \approx K_j\ \vec{y}) \\ \textsc{Match}((K_i\ \vec{p})\ \vec{q},\ \Gamma \vdash (K_j\ \vec{y})\ \vec{u} \rhd \Delta') \Downarrow \langle C_j, U_j, D_j \rangle \\ C = \bigcup C_j \qquad U = \bigcup U_j \qquad D = (\bigcup D) \cup \{\Gamma \vdash x\ \vec{u} \rhd \Delta \cup (x \approx \bot)\} \end{array}}{\textsc{Match}((K_i\ \vec{p})\ \vec{q},\ \Gamma \vdash x\ \vec{u} \rhd \Delta) \Downarrow \langle C, U, D \rangle}\ \textsc{ConVar}$$

Rule CConVar specializes variable pattern x to $K_i\ \vec{y}$ while Rule ConVar generates all refinements of x. As we discussed in Section 3.3.2, in the next iteration all K_j where $j \neq i$ will produce an empty covered set.

Though Section 3.3.2 stresses that Rule CConVar is more performant by generating only the refinement that will match, we need all refinements for computing the uncovered set; the benefits of merging the three functions outweigh the performance gain of simplifying the covered set computation.

It is straightforward to prove that the two formalizations of the algorithm behave identically:

Theorem 2 (Algorithm Equivalence)**.** *The following holds:*

$$\textsc{Match}(\vec{p},\ v) \Downarrow \langle C, U, D \rangle \iff (\mathcal{C}\ \vec{p}\ v = C) \wedge (\mathcal{U}\ \vec{p}\ v = U) \wedge (\mathcal{D}\ \vec{p}\ v = D)$$

The simultaneous computation of the covered, uncovered, and divergent sets has more benefits, as we illustrate below.

4.1.2 Guard Trees

As we briefly discussed in Section 3.2, the syntax specification allows only for a guard vector per clause. Though this design choice does not affect expressivity, it does have a negative impact on performance. For example, function abs_2 from Section 2.4.2 would give rise to two target clauses:

$$\begin{array}{lll} x & (\mathit{True} \leftarrow x < 0) & \rightarrow\ -x \\ y & (\mathit{True} \leftarrow y \geq 0) & \rightarrow\ y \end{array}$$

This approach is problematic for mainly two reasons:

- First, we need to replicate the pattern vector of each clause as many times as the number of guard vectors it is accompanied by. In turn, this means that the algorithm needs to traverse more pattern vectors. Since the algorithm has exponential worst-case behavior (see Section 3.4.5), the linear increase of pattern vectors may result in a significant performance penalty.
- The algorithm implicitly assumes that each pattern vector binds distinct variables; duplication of a pattern vector requires another phase of α-renaming of all variables the vector binds.

Thus, instead of replicating the pattern vector of each clause, we preserve the tree-like structure of the source language: the left-hand-side of each clause may be terminated either by no guards (ϵ), or a list of guard vectors $(\vec{p}_1, \ldots, \vec{p}_n)$.

The algorithm of Figure 4.1 can readily support the updated syntax with minimal changes. We only need to add one more rule to cover cases where the pattern vector is followed by a list of guard vectors:

$$\dfrac{\begin{array}{c} U_0 = \{\Gamma \vdash \epsilon \rhd \Delta\} \qquad \textsc{Match}(\vec{p}_i,\ U_{i-1}) \Downarrow \langle C_i, U_i, D_i \rangle \\ C = C_1 \cup \ldots \cup C_n \qquad U = U_n \qquad D = D_1 \cup \ldots \cup D_n \end{array}}{\textsc{Match}((\vec{p}_1, \ldots, \vec{p}_n),\ \Gamma \vdash \epsilon \rhd \Delta) \Downarrow \langle C, U, D \rangle}\ \textsc{GTree}$$

Rule GTree essentially traverses the guard vectors in a depth-first fashion: the set that remains uncovered by the first guard vector is passed as input to the second, etc. The covered and divergent sets for the whole clause are simply the union of the individual corresponding sets, and the uncovered set of the clause is the final uncovered set U_n.

Judgment $\textsc{Match}(\vec{p},\ S) \Downarrow \mathcal{R}$ generalizes judgment $\textsc{Match}(\vec{p},\ v) \Downarrow \mathcal{R}$ to operate on a value set abstraction S instead of single value vector abstraction v:

$$\frac{\begin{array}{c} i \in [1 \dots n] \ :\ \textsc{Match}(\vec{p},\ v_i) \Downarrow \langle C_i, U_i, D_i \rangle \\ C = C_1 \cup \dots \cup C_n \qquad U = U_1 \cup \dots \cup U_n \qquad D = D_1 \cup \dots \cup D_n \end{array}}{\textsc{Match}(\vec{p},\ \{v_1, \dots, v_n\}) \Downarrow \langle C, U, D \rangle}\ \textsc{ValSetAbs}$$

The alert reader will have noticed that the traversal pattern of Rule GTree is the same as of the general algorithm we presented in Figure 3.1. Indeed, as we discuss in Section 4.1.2 below, we can naturally generalize pattern matrices to *pattern trees* and open up new possibilities for further pattern matching extensions (and consequently pattern match checking).

4.1.3 Literal and Negative Literal Patterns

Finally, in order to achieve a competitive performance,[1] we extend the pattern language of Section 3.2.1 to include literal patterns l:

$$p, q \ ::= \ x \mid K\ \vec{p} \mid G \mid l \qquad \textit{target pattern}$$

Accordingly, we also extend the value abstractions presented in Section 3.3.1 to accommodate literal patterns:

$$u, w \ ::= \ x \mid K\ \vec{u} \mid l \mid x \notin \bar{l} \qquad \textit{value abstraction}$$

Additionally, we make use of *negative information* (Sestoft, 1996), via syntactic form $x \notin \bar{l}$. A value abstraction of the form $x \notin \bar{l}$ captures all literals x (of the appropriate type) that are not equal to any of the literals $\bar{l}$.

These changes introduce eight more cases in the algorithm of Section 4.1.

Literal Cases The first five cases are concerned with literal patterns alone and are presented in Figure 4.2.

Rules LitEq and LitNe behave similarly to Rules ConEq and ConNe: the former handles cases where the literal value abstraction matches the literal pattern and the latter cases where they differ.

Since value abstractions now involve negative literal information, we include Rules NLitNotIn and NLitIn.

[1]Though the translation of Section 3.2.2 is semantically sound, the lack of literal patterns induces a serious run-time overhead in pattern match checking involving literals. The introduction of negative literal patterns partially addresses this issue; the other measures we take are discussed in Section 4.3 below.

Figure 4.2 Pattern Match Checking (Literal Rules)

$$\frac{\textsc{Match}(\vec{p},\ \Gamma \vdash \vec{u} \triangleright \Delta) \Downarrow \mathcal{R}}{\textsc{Match}(l\ \vec{p},\ \Gamma \vdash l\ \vec{u} \triangleright \Delta) \Downarrow \mathit{map}_{\mathcal{R}}\ (\mathit{ucon}\ l)\ \mathcal{R}}\ \textsc{LitEq}$$

$$\frac{l_1 \neq l_2}{\textsc{Match}(l_1\ \vec{p},\ \Gamma \vdash l_2\ \vec{u} \triangleright \Delta) \Downarrow \langle \varnothing, \{\Gamma \vdash l_2\ \vec{u} \triangleright \Delta\}, \varnothing \rangle}\ \textsc{LitNe}$$

$$\frac{\begin{array}{c} l \notin \bar{l} \qquad U_1 = \{\ \Gamma \vdash (x \notin (l, \bar{l}))\ \vec{u} \triangleright \Delta\ \} \\ \textsc{Match}(l\ \vec{p},\ \Gamma \vdash l\ \vec{u} \triangleright (\Delta \cup x \approx l)) \Downarrow \langle C, U, D \rangle \end{array}}{\textsc{Match}(l\ \vec{p},\ \Gamma \vdash (x \notin \bar{l})\ \vec{u} \triangleright \Delta) \Downarrow \langle C, (U_1 \cup U), D \rangle}\ \textsc{NLitNotIn}$$

$$\frac{l \in \bar{l}}{\textsc{Match}(l\ \vec{p},\ \Gamma \vdash (x \notin \bar{l})\ \vec{u} \triangleright \Delta) \Downarrow \langle \varnothing, \{\Gamma \vdash (x \notin \bar{l})\ \vec{u} \triangleright \Delta\}, \varnothing \rangle}\ \textsc{NLitIn}$$

$$\frac{\begin{array}{c} U_1 = \{\Gamma \vdash (x \notin \{l\})\ \vec{u} \triangleright \Delta\ \} \qquad D_1 = \{\Gamma \vdash x\ \vec{u} \triangleright (\Delta \cup x \approx \bot)\ \} \\ \textsc{Match}(l\ \vec{p},\ \Gamma \vdash l\ \vec{u} \triangleright \Delta \cup x \approx l) \Downarrow \langle C, U, D \rangle \end{array}}{\textsc{Match}(l\ \vec{p},\ \Gamma \vdash x\ \vec{u} \triangleright \Delta) \Downarrow \langle C, (U_1 \cup U), (D_1 \cup D) \rangle}\ \textsc{LitVar}$$

Rule NLitNotIn covers cases where the pattern literal l is not included in the negative pattern $(x \notin \bar{l})$. Essentially, a value vector abstraction of the form $(\Gamma \vdash (x \notin \bar{l})\ \vec{u} \triangleright \Delta)$ represents a (possibly infinite) set of value vector abstractions:

$$\begin{array}{ll} \Gamma \vdash l_1\ \vec{u} \triangleright (\Delta \cup x \approx l_1) & \qquad (l_1 \notin \bar{l}) \\ \Gamma \vdash l_2\ \vec{u} \triangleright (\Delta \cup x \approx l_2) & \qquad (l_2 \notin \bar{l}) \\ \dots & \end{array}$$

Hence, there exists an l_i such that $l_i = l$. For that value vector abstraction matching may succeed, but all remaining value vector abstractions remain uncovered, which we capture in uncovered set U_1. The recursive call $\textsc{Match}(l\ \vec{p},\ \Gamma \vdash l\ \vec{u} \triangleright (\Delta, x \approx l)) \Downarrow \langle C, U, D \rangle$ deals with the matching case.

Rule NLitIn handles cases where $l \in \bar{l}$. Since pattern $(x \notin \bar{l})$ represents all cases where x *is not* any of the literals in $\bar{l}$, this is an obvious mismatch. Hence, the covered and divergent sets are empty, and the input uncovered value vector abstraction remains uncovered.

Notice that since a negative literal $(x \notin \bar{l})$ represents a (possibly infinite) set of literals, it is in WHNF. Hence, neither Rule NLitNotIn nor Rule NLitIn extends the divergent set.

Finally, Rule LitVar is equivalent to Rule ConVar, but for literal patterns: uncovered set U_1 captures all refinements that do not match literal pattern l, and set D_1 captures cases where evaluation diverges. The recursive call captures the only refinement of x that can still match:

$$\textsc{Match}(l\ \vec{p},\ \Gamma \vdash l\ \vec{u} \rhd \Delta \cup x \approx l) \Downarrow \langle C, U, D \rangle$$

Literal-Constructor Cases Additionally, we have three degenerate cases which are concerned with matches that mix overloaded and non-overloaded syntax. These cases handle examples like the one we discussed in Section 2.4.2 (adapted from GHC bug report #322)

The key technique in these rules is to abstract over a literal pattern l using a fresh term variable x (and record this in a term equality $(x \approx l)$, so that the algorithm can proceed using Rule ConVar/VarCon in the next iteration. Reasoning about equalities of the form $(\mathit{from}\ l \sim K\ \overline{e})$ is completely left to the term-oracle.

The rules are straightforward so we omit them for brevity. The interested reader can find them in the source code of GHC.[2]

Common Prefixes The introduction of negative literal patterns greatly improves the performance of the algorithm in cases where patterns with common prefixes are considered. Indeed, GHC bug report #11303 presents an example on which the naive implementation of the algorithm exhibits exponential behavior.

In essence, if we desugar literal patterns into guards (as we originally presented in Figure 3.3), then reasoning about literal patterns is entirely non-structural, and thus handled by the (possibly expensive) term oracle. Instead, explicit literal and negative literal patterns allow us to detect non-matching cases quickly, and avoid generating covered, uncovered, and divergent sets with unsatisfiable constraints.

4.2 Empty Case Expressions

A Haskell extension that we have not considered until now is that of empty case expressions (enabled in GHC by pragma `EmptyCase`). Such cases are often useful when dealing with GADTs.

[2]Cases ConLit, LitCon, and ConNLit, found in file `https://github.com/ghc/ghc/blob/master/compiler/deSugar/Check.hs`.

Empty Case Expressions For example, we can define finite sets *Fin* and safe lookup function *vlookup* over length-indexed vectors *Vec* (see Section 2.3) as follows:

$$\begin{array}{ll}
\textbf{data}\ \mathit{Fin} :: \mathit{Nat} \to \star\ \textbf{where} & \mathit{vlookup} :: \mathit{Fin}\ n \to \mathit{Vec}\ n\ a \to a \\
\quad \mathit{FZ} :: \mathit{Fin}\ (\mathit{Succ}\ n) & \mathit{vlookup}\ \mathit{FZ}\qquad (\mathit{VC}\ x\ xs) = x \\
\quad \mathit{FS} :: \mathit{Fin}\ n \to \mathit{Fin}\ (\mathit{Succ}\ n) & \mathit{vlookup}\ (\mathit{FS}\ n)\ (\mathit{VC}\ x\ xs) = \mathit{vlookup}\ n\ xs
\end{array}$$

Obviously, in the presence of GADTs it is very common to have uninhabited types: for example type (*Fin Zero*) is uninhabited (under call-by-value semantics; under call-by-name all types are inhabited by $\perp$). To explicitly mark such cases, Agda for example introduces the *absurd pattern* (Norell, 2009). In Haskell, we can achieve the same effect by means of empty case expressions. For example:

$$\begin{array}{l}
\mathit{absurd} :: \mathit{Fin}\ \mathit{Zero} \to ()\\
\mathit{absurd}\ x = \textbf{case}\ x\ \textbf{of}\ \{\}
\end{array}$$

Under lazy semantics, function *absurd* is non-exhaustive: $\perp$ inhabits type (*Fin Zero*) so the function call (*absurd* $\perp$) would crash due to non-exhaustiveness, rather than due to evaluating the diverging argument. By design though, empty case expressions in Haskell have strict semantics (lazy semantics would defeat the purpose of introducing empty case expressions in the first place): one could simply rewrite the above to the following:

$$\begin{array}{l}
\mathit{absurd} :: \mathit{Fin}\ \mathit{Zero} \to ()\\
\mathit{absurd}\ x = \mathit{seq}\ x\ (\textbf{case}\ x\ \textbf{of}\ \{\})
\end{array}$$

(that is, evaluate the argument to WHNF and then match against no patterns).

Unhandled Empty Case Expressions Obviously, the algorithm of Section 3.3.2 does not take this into account: in case the set of clauses of a match is empty, the original uncovered set is returned and the match is considered unconditionally non-exhaustive. This behavior is captured in GHC's bug report #10746.

This is not new, Maranget (2007) similarly assumes that all types are inhabited, thus exhibiting the same behavior. Also, notice that the match in function *absurd* is exhaustive, but the following is not:

$$\begin{array}{l}
\mathit{silly} :: \mathit{Int} \to ()\\
\mathit{silly}\ x = \textbf{case}\ x\ \textbf{of}\ \{\}
\end{array}$$

This is due to type *Int* being inhabited: a call (*silly* 42) would crash at run-time with a non-exhaustiveness error.

Solution: Inhabitation Checking What all the above point at is that coverage checking for pattern matching with strict semantics is related to *inhabitation checking* (whether a type is inhabited by terms or not). Indeed, as Garrigue and Normand (2015) recently pointed out, the two are equivalent. Unfortunately, inhabitation checking is undecidable (Urzyczyn, 1997), but we can at least deal with the most common cases. This is the approach we took for GHC.

Challenges and Design Choices To summarize, an empty case (with strict semantics) is exhaustive iff the type of the scrutinee is uninhabited. In order to issue more informative warnings, instead of a simple yes/no, our implementation also generates the matches that are considered missing. In order to generate such matches, we need to address the following three challenges:

1. *Type Family Redexes.* First, type family applications may complicate inhabitation checking, since they essentially introduce type-level expressions. For example, type expression $F\ Int$ is not necessarily inhabited: if there is an instance of the form $F\ Int \sim Bool$, we can normalize/reduce $F\ Int$ to $Bool$, to expose that it is in fact an inhabited closed algebraic data type.

2. *Newtypes.* The Haskell standard (Peyton Jones, 2003) introduces the so-called *newtype*s. A newtype declaration introduces a new type, which is isomorphic to an existing one. This allows the separation of the two in the source language (e.g., *Length* vs. *Width*), while preserving an efficient run-time representation for both (e.g., they can both be represented by an *Integer*).

   Newtypes complicate matters. Consider the following example:

   $$\begin{array}{llll} \textbf{data} & Foo1 = Foo1\ Int & & y_1 = \textbf{case}\ \bot\ \textbf{of}\ \{\ Foo1\ x \to 1\ \} \\ \textbf{newtype} & Foo2 = Foo2\ Int & & y_2 = \textbf{case}\ \bot\ \textbf{of}\ \{\ Foo2\ x \to 2\ \} \end{array}$$

   y_1 evaluates to $\bot$, as we have illustrated in detail in Chapters 2 and 3. The latter though, evaluates to 2: newtype constructor *Foo2* does not force any evaluation. This would be more obvious if we have (equivalently) defined y_2 as

   $$y_2 = \textbf{case}\ \bot\ \textbf{of}\ \{\ x' \to 2\ \}$$

   To summarize, in order to check for inhabitation in the presence of newtypes, we need to reason about the underlying type (*Int* in this example), not the source type. Yet, the equivalence between the newtype and its underlying representation needs to be accounted for when issuing warnings: 3 has type *Int*, not *Foo2*.

3. *Data Family Redexes.* GHC also extends Haskell with another variant of datatype declarations, known as *data families* (Chakravarty et al., 2005b).

Data families differ from ordinary data types in that they represent an open, indexed family of types. For example, one can abstract over monads with support for mutable references as follows:[3]

```
class Monad m ⇒ RefM m where
  data Ref m v
  newRef :: v → m (Ref m v)
  ...
```

and make both *IO* and *ST* monads instances of class *RefM* as follows:

```
instance RefM IO where
  data Ref IO v = RefIO (IORef v)
  newRef v = fmap RefIO (newIORef v)
  ...

instance RefM (ST s) where
  data Ref (ST s) v = RefST (STRef s v)
  newRef v = fmap RefST (newSTRef v)
  ...
```

Function *newRef* has the generic type

$$\forall m.\ \forall v.\ \mathit{RefM}\ m \Rightarrow v \rightarrow m\ (\mathit{Ref}\ m\ v)$$

Data families are very useful in practice, but they also pose a challenge for pattern match checking. Internally, the above data family instances are elaborated into two separate datatype declarations, along with two *equality axioms* (Sulzmann et al., 2007a):

```
data Ref s v

data RRefIO v = RefIO (IORef v)
data RRefST s v = RefST (STRef s v)

axiom ax_1 v : Ref IO v ∼ RRefIO v
axiom ax_2 s v : Ref (ST s) v ∼ RRefST s v
```

Thus, source appeals to the abstract type (*Ref s v*) are translated into GHC's intermediate language as explicit casts of the specialized data types, using axioms ax_1 and ax_2.

Type constructors *RRefIO* and *RRefST* are called *representation constructors*. For pattern match checking we reason about representation constructors, but the warnings issued to the programmer need to refer to the source abstract type.

[3]Example borrowed from the original work of Chakravarty et al. (2005b) on data families.

Implementation All the above are handled by function

$$\begin{array}{l} \mathit{pmTopNormaliseType\_maybe} \\ \quad :: \mathit{FamInstEnvs} \rightarrow \mathit{Type} \rightarrow \mathit{Maybe}\ (\mathit{Type}, \mathit{Type}, [\mathit{DataCon}]) \end{array}$$

Given all top-level type family axioms *FamInstEnvs* and the type of the scrutinee, we generate three entities:

(a) A normalized type, which is equal to the type of the scrutinee in source Haskell: in this type newtypes and data family redexes are not normalized.

(b) The actual normalized type which is not necessarily equal to the input type in source Haskell: we reason about the match using the underlying type but use the other two results to issue warnings that respect newtypes and do not show the representation constructors.

(c) A list of all newtype data constructors, each one corresponding to a newtype rewrite performed while computing type (b).

Next, we introduce the function

$$\begin{array}{l} \mathit{inhabitationCandidates} :: \mathit{FamInstEnvs} \rightarrow \mathit{Type} \\ \quad \rightarrow \mathit{PmM}\ (\mathit{Either}\ \mathit{Type}\ [(\mathit{ValAbs}, \mathit{ComplexEq}, \mathit{Bag}\ \mathit{EvVar})]) \end{array}$$

Function *inhabitationCandidates* utilizes *pmTopNormaliseType_maybe* to generate one of the following results:

- *Left ty*, in case the input type cannot be reduced to a closed algebraic type (or if it is one trivially inhabited, like *Int*), or
- *Right candidates*, if reduction to a closed algebraic datatype is possible. The list *candidates* contains all possible refinements of the appropriate type, accompanied by the term- and type-constraints they give rise to.

Last, function $\mathit{checkEmptyCase}' :: \mathit{Id} \rightarrow \mathit{PmM}\ \mathit{PmResult}$ ties them all together: by means of function *inhabitationCandidates*, it generates and filters the possible inhabitation candidates that come with unsatisfiable constraints.

The implementation of this extension can be found in GHC or the following differential revision:

`https://phabricator.haskell.org/D2105`

Example To illustrate how all the above cases come into play, consider the following definitions:

$$\begin{array}{l}
\textbf{data family } T\ a \\
\textbf{data instance } T\ \mathit{Int} = \mathit{T1} \mid \mathit{T2}\ \mathit{Bool} \\
\\
\textbf{newtype } \mathit{G1} = \mathit{MkG1}\ (T\ \mathit{Int}) \\
\textbf{newtype } \mathit{G2} = \mathit{MkG2}\ \mathit{G1} \\
\\
\textbf{type family } F\ a \\
\textbf{type instance } F\ \mathit{Int} \quad = F\ \mathit{Char} \\
\textbf{type instance } F\ \mathit{Char} = \mathit{G2}
\end{array}$$

In order to check expression ($\textbf{case}\ (x :: F\ \mathit{Int})\ \textbf{of}\ \{\}$), we proceed as follows:

1. First, the data family declarations give rise to an abstract type $T\ a$, a (representation) data type RTInt, and an equality axiom stating that $T\ \mathit{Int}$ is equal to RTInt:

$$\begin{array}{l}
\textbf{data } T\ a \\
\textbf{data } \mathit{RTInt} = \mathit{T1} \mid \mathit{T2}\ \mathit{Bool} \\
\textbf{axiom } \mathit{ax} : T\ \mathit{Int} \sim \mathit{RTInt}
\end{array}$$

2. Second, we normalize type ($F\ \mathit{Int}$) via function *pmTopNormaliseType_maybe* to compute (a) the normalized type $\mathit{G2}$, which is equal to $F\ \mathit{Int}$ in the source language, (b) the normalized type RTInt, which is equivalent to $F\ \mathit{Int}$ in the target language but not source Haskell, and (c) the list of newtype data constructors $[\mathit{MkG2}, \mathit{MkG1}]$: for any y of type RTInt we have that $(\mathit{MkG2}\ (\mathit{MkG1}\ y) :: F\ \mathit{Int})$.

3. Third, function *inhabitationCandidates* generates all possible refinements of y: $\mathit{T1}$ and $\mathit{T2}\ z$.

4. Last, we filter out the ones for which the accompanying constraints are unsatisfied (in this case both patterns are well-typed), to emit a warning about the missing patterns:

$$\begin{array}{l}
\mathit{MkG2}\ (\mathit{MkG1}\ \mathit{T1}) \\
\mathit{MkG2}\ (\mathit{MkG1}\ (\mathit{T2}\ z))
\end{array}$$

   Notice that if $T\ \mathit{Int}$ was a GADT, some of the cases could have been ill-typed and thus removed.

Figure 4.3 Specialized Clause Processing (Rule CConVar)

$$\mathcal{C} \quad ((K_i\ \vec{p})\ \vec{q}) \quad (\Gamma \vdash x\ \vec{u} \rhd \Delta) \quad = \quad \mathcal{C}\ ((K_i\ \vec{p})\ \vec{q})\ (\Gamma' \vdash (K_i\ \vec{y})\ \vec{u} \rhd \Delta')$$

$$\text{where} \quad \vec{y}\#\Gamma \quad \vec{b}\#\Gamma \quad (x : T\ \vec{\tau}_x) \in \Gamma \quad K_i :: \forall \vec{a}.\ \forall \vec{b}.\ Q \Rightarrow \vec{\tau} \rightarrow T\ \vec{a}$$
$$\theta = [\vec{\tau}_x/\vec{a}] \quad \Gamma' = \Gamma, \vec{b}, \vec{y} : \theta(\vec{\tau}) \quad \Delta' = \Delta \cup \theta(Q) \cup x \approx K_i\ \vec{y}$$

4.3 Performance Improvements

As we mentioned in Section 4.1, we have optimized the presentation of our algorithm in Section 3.3.2 for clarity, rather than run-time performance. Even though we cannot improve upon the asymptotic worst-case time complexity, various measures can improve the average performance a big deal.

The all-in-one algorithm of Figure 4.1 constitutes only a small part of the possible improvements we can implement. In this section we discuss several optimizations we have implemented to improve the performance of the algorithm on the most common cases.

4.3.1 Implicit Solving

The formulation of the algorithm in both Sections 3.3.2 and 4.1.1 generates type constraints for the oracle with a high frequency. For instance, Rule CConVar of the $\mathcal{C}$ function generates a new type equality constraint $\tau \sim \tau_x$ every time it fires, even for Haskell '98 data types.

While there are good reasons for generating these constraints in general, we can in many cases avoid generating them explicitly and passing them on to the oracle. Instead, we can handle them immediately and much more cheaply. One important such case is covered by the specialized variant of rule CConVar in Figure 4.3: the type τ_x has the form $T\ \vec{\tau}_x$, where T is also the type constructor of the constructor K_i. This means that the generated type constraint $\tau \sim \tau_x$ actually has the form $T\ \vec{a} \sim T\ \vec{\tau}_x$. We can simplify this constraint in two steps. Firstly, we can decompose it into simpler type equality constraints $a_i \sim \tau_{x,i}$, one for each of the type parameters. Secondly, since all type variables $\vec{a}$ are actually fresh, we can immediately *solve* these constraints by substituting all occurrences of $\vec{a}$ by $\vec{\tau}_x$. Rule CConVar incorporates this simplification and does not generate any type constraints at all for Haskell '98 data types.

Since the same applies to the computation of the uncovered and divergent sets, we incorporate this simplification in the implementation of Rule ConVar (Figure 4.1.1).

4.3.2 Incremental Solving

Many constraint solvers, including the OutsideIn(X) solver (Schrijvers et al., 2009; Vytiniotis et al., 2011), support an incremental interface:

$$\mathit{solve} :: \mathit{Constraint} \to \mathit{State} \to \mathit{Maybe}\ \mathit{State}$$

In the process of checking given constraints C_0 for satisfiability, they also normalize them into a compact representation. When the solver believes the constraints are satisfiable, it returns their normal form: a *state* σ_0. When later the conjunction $C_0 \wedge C_1$ needs to be checked, we can instead pass the state σ_0 together with C_1 to the solver. Because σ_0 has already been normalized, the solver can process the latter combination much more cheaply than the former.

It is very attractive for our algorithm to incorporate this incremental approach, replace the constraints Δ by normalized solver states σ and immediately solve new constraints when they are generated. Because the algorithm refines step by step one initial value abstraction into many different ones, most value abstractions share a common prefix of constraints. By using solver states for these common prefixes, we share the solving effort among all refinements and greatly save on solving time. Moreover, by finding inconsistencies early, we can prune eagerly and avoid refining in the first place.

In fact, our current oracle implementation incorporates only partially this optimization; sharing of the oracle state is provided for term constraints only. Section 4.4 below discusses the implementation of the oracle at length.

4.3.3 General Term Equalities

If we faithfully follow the translation of literal patterns of Figure 3.3, a literal l should be translated (in two steps) into the following pattern vector:

$$x\ (\mathit{True} \leftarrow (x == l))$$

Hence, by applying Rule UGuard on the second pattern, constraint set Δ will contain constraints $(y \sim \mathit{False})$ and $(y \sim (x == l))$. Since y is a fresh term variable generated by Rule UGuard to capture the result of expression $(x == l)$, it can be eliminated; the above two constraints can be replaced by $\mathit{False} \sim (x == l)$.

Though this seemed like a needless optimization at the first stages of our development, bug reports #11160 and #11161 illustrated that the size of constraint sets can significantly increase in matches against literal patterns. More specifically, #11160 includes the following function:

$$\begin{array}{lcl} \mathit{foo} :: \mathit{Int} \to \mathit{Int} & & \\ \mathit{foo}\ 1 & = & 0 \\ \mathit{foo}\ 2 & = & 1 \\ \ldots & & \\ \mathit{foo}\ 5000 & = & 4999 \\ \mathit{foo}\ n & = & n + 1 \end{array}$$

With the naive translation, checking function *foo* generates twice as many term equalities as needed (in this case this amounts to 10000 term equalities). Since our term oracle is (at least) quadratic in the number of constraints, this led to unacceptable compilation times (see #11160).

Thus, we redesigned the oracle to manipulate general term equalities of the form $e_1 \sim e_2$, instead of the more restrictive form $x \sim e$.

4.3.4 Overloaded Literals

Similarly to simple literals l, overloaded literals ol can also introduce a severe performance penalty if translated naively. As we illustrated in Section 3.2.2, an overloaded literal pattern ol represents a function application (denoted as $(\mathit{from}\ ol)$ in Figure 3.3).

Indeed, if a class instance of the form

$$\begin{array}{c} \textbf{instance}\ \mathit{Num}\ a \Rightarrow \mathit{Num}\ (\mathit{Maybe}\ a)\ \textbf{where} \\ \ldots \\ \mathit{fromInteger}\ i = \mathit{Just}\ (\mathit{fromInteger}\ i) \end{array}$$

is available, one can write the following function:

$$\begin{array}{lcl} f :: \forall a.\ \forall b.\ (\mathit{Eq}\ a, \mathit{Num}\ a, \mathit{Num}\ b) \Rightarrow \mathit{Maybe}\ a \to b & & \\ f\ (\mathit{Just}\ 4) & = & 1 \\ f\ 5 & = & 2 \end{array}$$

According to the translation given in Figure 3.3, the second clause is desugared into the following:

$$x\ (\mathit{True} \leftarrow x == \mathit{fromInteger}\ 5) \to 2$$

Obviously, for the algorithm to process this clause with precision, the term oracle would have to inline the definition of function $\mathit{fromInteger} :: \mathit{Num}\ a \Rightarrow$

Integer $\rightarrow a$. As we discuss in Section 4.4.2 below, our current term oracle does not inline function definitions so this approach would give very poor results for overloaded literals. Furthermore, all generated constraints would be "dormant" (and hence needlessly burden the term oracle), since the term oracle cannot handle them.

We simplify matters by treating overloaded literals like simple literals: two overloaded literals that *look* different are considered different. Of course, this oversimplification is not faithful to their semantics. Consider for example the following program:

$$\begin{array}{ll}
\textbf{instance}\ \mathit{Num}\ \mathit{Bool}\ \textbf{where} & f :: \mathit{Bool} \rightarrow () \\
\quad \ldots & f\ 1 = () \\
\quad \mathit{fromInteger}\ \_ = \mathit{False} & f\ 2 = ()
\end{array}$$

With our current implementation, a warning of the following form is issued:

```
Pattern match(es) are non-exhaustive in an equation for ‘f’:
    Patterns not matched: x with x not one of [1,2]
```

which is incorrect. To illustrate why, let us consider how function f is elaborated internally. The definition of f is roughly equivalent to the following:

$$\begin{array}{lllll}
f\ x & \mid & x == \mathit{fromInteger}\ 1 & = & () \\
 & \mid & x == \mathit{fromInteger}\ 2 & = & ()
\end{array}$$

Since ($\mathit{fromInteger}\ e$) reduces to *False* for any expression e, the second clause is redundant; in fact any other numeric literal pattern would be redundant too. Moreover, though a clause ($f\ 3 = \ldots$) is not provided, a call ($f\ 3$) (which is equivalent to f ($\mathit{fromInteger}\ 3$)) results in $()$, not a non-exhaustiveness error. The only ways to make function f exhaustive are either to add a catch-all pattern x, or explicitly match against *True*.

The bottom line is this: reasoning about overloaded literals becomes too complicated if we treat them faithfully to their specification. Thus, we avoid such complexity by deliberately treating them as non-overloaded literal patterns. This approach was taken by GHC before the development of our algorithm and therefore Haskell users are familiar with its behavior.

4.3.5 Representation of Covered and Divergent Sets

Though both formalizations of the algorithm we presented (Sections 3.3.2 and 4.1.1) compute all three sets, only one of them is actually of importance to the user: the uncovered set U. The other two (covered sets C and divergent sets D) are only of importance for answering the question "is this clause useful, redundant, or useful only for evaluation?".

Hence, our implementation does not keep track of the covered and divergent sets. Instead, our representation of triples $\mathcal{R}$ looks more like the following:

$$\mathcal{R} ::= \langle \mathit{Bool}, U, \mathit{Bool} \rangle$$

This design choice significantly reduces both memory consumption and processing time: the sets are neither stored, nor processed.

4.3.6 Eager Solving

Even after the implementation of all the aforementioned optimizations, our original implementation still performed poorly in several ordinary examples. The source of this slowdown turned out to be the exponential size of the uncovered sets we generated.

Indeed, a closer look at the definition of *patVecProc* shows that the naive implementation of the algorithm implements a generation phase first, followed by a filtering phase. Instead, we have optimized our implementation to employ more frequent calls to the oracles, so that the generated sets always contain denotationally non-empty value vector abstractions.

This optimization significantly improved the performance of the algorithm in the average case, and addressed GHC bug report #11374, among others.

4.3.7 Pattern Coercions

As we discussed in passing in Section 4.2, GHC's intermediate language allows explicit type casts, using type-equality witnesses (e.g., axioms ax_1 and ax_2 in the previous section). This feature allows the encoding of source features that introduce implicit type equalities, like GADTs, type families, and data families.

During type inference, proofs for implicit source-level casts are constructed, and the syntax tree is richly decorated with type information and explicit casts. Thus, the language of source patterns—as implemented in GHC—includes *pattern coercions*. Given an explicit proof γ of type-equality $\tau_1 \sim \tau_2$ and a pattern p of type τ_2, a pattern coercion (p, γ) represents a match of the form

$$\textbf{case}\ (x \triangleright \gamma)\ \textbf{of}\ \{\ p \to \ldots\ \}$$

where $\triangleright$ performs an explicit type cast. Hence, the type of the argument matched against the pattern is of type τ_1. In order to handle pattern coercions, we extended the desugaring algorithm of Figure 3.3 to handle pattern coercions in a type-preserving manner:

$$\mathit{translate}_p((p, \gamma)) = x\ (\mathit{translate}_p(p) \leftarrow x \triangleright \gamma)$$

Unfortunately, this translation gives rise to guard patterns for programs that use data families (the source type constructor needs to be cast to the representation constructor), inducing a compilation slowdown. This issue is captured in GHC bug report #11276.

Though this issue cannot be addressed in the general case (in a type-preserving manner), we optimized our translation algorithm to omit explicit casts when they are trivial reflexivity proofs. This minor change in combination with the eager solving strategy we presented in Section 4.3.6 and the incremental nature of our term oracle (see Section 4.4 below), sufficiently addressed this performance issue.

Summary To summarize, the algorithm we developed in Chapter 3 required several modifications for its implementation in GHC to have a competitive performance. Fortunately, all optimizations were straightforward and easy to implement. Most of the optimizations we discussed in this section were part of the following revision:

`https://phabricator.haskell.org/D1795`

One important takeaway from this section is that pattern match checking of structural pattern matching is significantly more efficient than checking of non-structural pattern matching: non-structural constructs give rise to guards, whose satisfiability we can only check by appeals to the term oracle.

This situation can be remedied if the oracle provides an incremental interface (making the calls cheaper) and is called frequently (thus eagerly pruning denotationally empty value vector abstractions from the generated sets).

4.4 The Oracle

As we mentioned in Section 3.1.3, the oracle judgment $\vdash_{\text{SAT}}$ is treated as a black box by the algorithm of Chapter 3. As long as it is conservative, any definition will do, even accepting all constraints. Our implementation does quite a bit better than that. In this section we present the details of the oracle (for both type- and term-level constraints) we use in our implementation.

4.4.1 Type-level Constraints

For type constraints we simply re-use the powerful type-constraint solver that GHC uses for type inference (Vytiniotis et al., 2011): OutsideIn(X). Hence, inconsistency of type constraints is defined uniformly and our oracle adapts automatically to any changes in the type system, such as type-level functions, type-level arithmetic, and so on.

More specifically, the interface with the type-constraint solver is implemented by function *tyOracle*, which has the following signature:

$$\mathit{tyOracle} :: \mathit{Bag}\ \mathit{EvVar} \to \mathit{PmM}\ \mathit{Bool}$$

In short, function *tyOracle* takes a set of constraints (*Bag EvVar*), and returns a boolean (*True* if the constraints are satisfiable and *False* otherwise). The result is wrapped in the *PmM* monad, which adds essential functionality like looking up the types of data constructors, generating fresh names, etc.

4.4.2 Term-level Constraints

For the handling of term-level constraints, we have implemented a vestigial satisfiability checker. Essentially, the term-oracle employs a variation of the standard unification algorithm originally introduced Robinson (Robinson, 1965; Martelli and Montanari, 1982)).

The oracle provides an incremental interface with the following signature

$$\mathit{tmOracle} :: \mathit{TmState} \to [\mathit{ComplexEq}] \to \mathit{Maybe}\ \mathit{TmState}$$

where *TmState* is the type of the term oracle state and *ComplexEq* is a type synonym, capturing equalities between Haskell terms:

$$\mathbf{type}\ \mathit{ComplexEq} = (\mathit{PmExpr}, \mathit{PmExpr})$$

Expressions *PmExpr* denote *lifted* Haskell expressions:

$$e' \ ::= \ x \mid K\ \overline{e'} \mid l \mid ol \mid e'_1 \approx e'_2 \mid \{e\} \qquad\qquad \textit{lifted expressions}$$

The syntax of lifted expressions e' separates the forms the solver can reason about from the expressions it does not handle. The latter are captured in syntactic form $\{e\}$.

Thus, only equalities with syntactically different sides are flagged as inconsistent (e.g., $\mathit{True} \approx \mathit{False}$). This enables us to see that abs_1 (Section 2.4.2) is exhaustive, but not abs_2. There is therefore plenty of scope for improvement, and various powerful term-level solvers, such as Zeno (Sonnex et al., 2012) and HipSpec (Claessen et al., 2013), could be used to serve as the oracle. Indeed, as we discussed in Section 3.5.7, the tool developed by Pavel Kalvoda and Tom Sydney Kerckhove already investigates the incorporation of the Z3 SMT solver.

One important weakness of the current term-oracle is that it does not handle applications (they are included in syntactic form $\{e\}$). In turn, this means that the checker currently cannot reason about view patterns. The reason behind this design choice is that reasoning about applications requires (a) inlining of function definitions, and (b) partial evaluation. Though both can be implemented, they can have a significant impact on performance. Furthermore, we have not identified a good criterion (easy to specify from the user's perspective) for ensuring termination of partial evaluation. Thus, such an extension constitutes part of our future work, once GHC users and developers reach a consensus about the desired expressivity/performance ratio for pattern match checking.

4.5 Evaluation

Our new pattern checker addresses the three challenges laid out in Section 2.4: GADTs, laziness, and guards. However in our evaluation, only the first turned out to be significant. Concerning laziness, none of our test programs triggered the warning for a clause that is irredundant, but has an inaccessible right-hand side; clearly such cases are rare! Concerning guards, our prototype implementation only has a vestigial term-equality solver, so until we improve it we cannot expect to see gains.

For GADT-rich programs, however, we do hope to see improvements. However, many programs do not use GADTs at all; and those that do often need to match against all constructors of the type anyway. So we sought test cases by asking the Haskell `libraries` list for cases where the authors missed accurate warnings for GADT-using programs. This has resulted in identifying 9 hackage packages and 3 additional libraries, available on GitHub.[4]

We compared three checkers. The baseline is, of course, vanilla GHC. However, GHC already embodies an *ad hoc* hack to improve warning reports for GADTs,

[4]https://github.com/amosr/merges/blob/master/stash/Lists.hs
https://github.com/gkaracha/gadtpm-example
https://github.com/jstolarek/dep-typed-wbl-heaps-hs

		GHC-1		**GHC-2**		**New**	
Hackage Packages	**LoC**	**M**	**R**	**M**	**R**	**M**	**R**
accelerate	11,393	11	0	9	0	8	14
ad	1,903	2	0	0	0	0	6
boolsimplifier	256	10	0	0	0	0	0
d-bus	2,753	45	0	42	0	16	1
generics-sop	1,008	0	0	0	0	0	3
hoopl	2,147	33	0	0	0	0	3
json-sop	393	0	0	0	0	0	2
lens-sop	280	2	0	0	0	0	2
pretty-sop	27	0	0	0	0	0	1
Additional tests	**LoC**	**M**	**R**	**M**	**R**	**M**	**R**
lists	66	1	0	0	0	0	3
heterogeneous lists	38	0	0	0	0	0	2
heaps	540	3	0	0	0	0	1

Table 4.1: Results

so we ran GHC two ways: both with (GHC-2) and without (GHC-1) the hack. Doing so gives a sense of how effective the *ad hoc* approach was compared with our new checker.

For each compiler we measured:

- *The number of missing clauses (M).* The baseline compiler GHC-1 is conservative, and reports too many missing clauses; so a lower M represents more accurate reporting.
- *The number of redundant (R) clauses.* The baseline compiler GHC-1 is conservative, and reports too few redundant clauses; so a higher R represents more accurate reporting.

The results are presented in Table 4.1. They clearly show that the ad-hoc hack of GHC-2 was quite successful at eliminating unnecessary missing pattern warnings, but is entirely unable to identify redundant clauses. The latter is where our algorithm shines: it identifies 38 pattern matches with redundant clauses, all of them catch-all cases added to suppress erroneous warnings. We also see a good reduction (-27) of the unnecessary missing pattern warnings. The remaining spurious missing pattern warnings in `accelerate` and `d-bus` involve pattern guards and view patterns; these can be eliminated by upgrading the term-level reasoning of the oracle.

Erroneous Suppression of Warnings We have found three cases where the programmer has erroneously added clauses to suppress warnings. We have paraphrased one such example in terms of the *Vec n a* type of Section 2.3.

```
data EQ n m where
  EQ :: n ~ m ⇒ EQ n m

eq :: Vec n a → Vec m a → EQ n m → Bool
eq VN        VN        EQ = True
eq (VC x xs) (VC y ys) EQ = x == y && eq xs ys
eq VN        (VC _ _)  _  = error "redundant"
eq (VC _ _)  VN        _  = error "redundant"
```

This example uses the *EQ n m* type as a witness for the type-level equality of n and m. This equality is exposed by pattern matching on *EQ*. Hence, the third and fourth clauses must be redundant. After all, we cannot possibly have an equality witness for $Zero \sim Succ\ n$. Yes, we can: that witness is $\bot :: EQ\ Zero\ (Succ\ n)$ and it is not ruled out by the previous clauses. Indeed, calls of the form $eq\ VN\ (VC\ x\ xs)\ \bot$ and $eq\ (VC\ x\ xs)\ VN\ \bot$ are not covered by the first two clauses and hence rightly reported missing. The bottoms can be flushed out by moving the equality witness to the front of the argument list and matching on it first. Then the first two clauses suffice.

GHC Tickets With the initial implementation of our new algorithm we have also been able to close nine GHC tickets related to GADT pattern matching (#3927, #4139, #6124, #8970) and literal patterns (#322, #2204, #5724, #8016, #8853).

During subsequent development more bug reports and feature requests were introduced and/or addressed (like the ones we discussed in Section 4.3). A comprehensive list of all the bug reports concerning the current implementation of the algorithm in GHC can be found in the corresponding GHC Wiki page:

`https://ghc.haskell.org/trac/ghc/wiki/PatternMatchCheck`

4.5.1 Performance Evaluation

As we discussed in Section 3.4.5, our algorithm has exponential behavior in the worst case. Nevertheless, we expect this behavior to be rare in practice. To confirm this expectation, we put our initial implementation (March 2015) to the test by collecting statistics concerning the size of sets C and U our algorithm generates for the packages of Table 4.1:

Maximum size of C/U	**Pattern Matches**	**(%)**
$1-9$	8702	97.90%
$10-99$	181	2.04%
$100-2813$	5	0.06%

Since there was significant variance in the results, we divided them into three size groups. Out of 8888 pattern matches checked in total, almost 98% of the generated and processed sets have a size less than 10. In fact, the vast majority (over 95%) have size 1 or 2.

The percentage of sets with size between 10 and 99 is 2.04%. We believe that this percentage is acceptable for types with many constructors and for pattern matches with many arguments.

Last but not least, we encountered 5 cases (only 0.06%) with extremely large sets ($\geq$ 100 elements). All of them were found in a specific library[5] of package `ad`. As expected, all these involved pattern matches had the structure of function f from Section 3.4.5:

$$\begin{array}{l} \textbf{data}\ T = A \mid B \mid C \\ f\ \ A\ \ A\ \ =\ \ \mathit{True} \\ f\ \ B\ \ B\ \ =\ \ \mathit{True} \\ f\ \ C\ \ C\ \ =\ \ \mathit{True} \end{array}$$

Notably, the most extreme example which generated an uncovered set of size 2813, matches against two arguments of type T with 54 data constructors (55 including $\perp$), a match that gives rise to 3025 different value combinations!

4.6 Scientific Output

This chapter has discussed several aspects of the pattern match checking algorithm implementation in GHC. The performance evaluation of Section 4.5.1 reflects the state of the implementation at the time of publication, while the rest of the sections present material we obtained from subsequent development.

Our pattern match checker has been part of the GHC codebase and thus available to GHC users since version 8.0.1.[6] This work restores the usefulness of

[5] Library Data.Array.Accelerate.Analysis.Match.

[6] `https://downloads.haskell.org/~ghc/8.0.1/docs/html/users_guide/8.0.1-notes.html`.

pattern match checking in GHC, an issue that has been open for over a decade (see bug report #595). In addition to restoring the developers' faith in the pattern match checker, the implementation of our algorithm promotes the use of GADTs at large, with the compiler providing more accurate warnings for programs using GADTs and other features. Finally, both the algorithm and its implementation are easily extensible, a trait that can be put to good use in the future when more extensions are implemented.

Part II

Type Classes

Chapter 5

Background

We now turn to the second part of this thesis, which is concerned with three extensions to Haskell's type class system that we have developed. Before we present the extensions in the forthcoming chapters, in this chapter we cover the necessary background. The chapter is structured as follows:

Section 5.1 discusses the notion of *polymorphism* and the various forms it comes in. Section 5.2 focuses on *parametric* polymorphism, and elaborates on the polymorphic lambda calculus, also known as System F. Section 5.3 explains the notions of *type reconstruction* (also known as *type inference*) and *elaboration*, by means of the Hindley-Damas-Milner system. Finally, Section 5.4 gives an informal overview of type classes, as presented in the literature; all formal aspects of type classes are deferred until Chapter 6.

5.1 Polymorphism

One of the most important concepts in programming languages is that of *polymorphism*. Polymorphism encapsulates the ability of a single interface to be used for entities of different types, thus, allowing for *code reusability*. In turn, reusability means more concise and safer code; the programmer needs to implement a function correctly *only once*.

Polymorphism comes in various forms; below we briefly discuss those forms that are most relevant for this thesis. A more detailed overview of the most common kinds of polymorphism is given by Pierce (2002, Chapter 23.2).

5.1.1 Parametric Polymorphism

Parametric polymorphism (also known as *genericity*) refers to the ability to write functions generically, so that values can be handled uniformly, independently of their type.

More specifically, a polymorphic function is said to be *parametric* over the type a of an argument if it behaves uniformly, independently of the instantiation of a. This is illustrated in the definition of the *reverse* function in Haskell, which reverses a list:

$$\begin{aligned} &\mathit{reverse} :: \forall a.\ [a] \to [a] \\ &\mathit{reverse}\ \ [] &&= [] \\ &\mathit{reverse}\ \ (x : \mathit{xs}) &&= \mathit{reverse}\ \mathit{xs} \ \text{++}\ [x] \end{aligned}$$

The type of *reverse* indicates that for *any* type a, given a list of elements of type a, it computes a list containing elements of type a. Notice how the definition of *reverse* does not inspect the elements x, but merely pattern matches against the structure of the list. This is precisely what allows *reverse* to work on lists containing elements of *any* type a:

```
Main> reverse [1,2,3]
[3,2,1]

Main> reverse ["a","b","c","d"]
["d","c","b","a"]
```

In the first example, *reverse* is applied to a list of *Integer*s while in the second to a list of *String*s.

In general, parametrically polymorphic functions act in the same way, independently of their instantiation. This property is known as the *"abstraction theorem"* (Reynolds, 1983) or the *"parametricity theorem"* (Wadler, 1989). Two highly relevant systems have this property: System F (Girard, 1972; Reynolds, 1974, 1983) and the Hindley-Damas-Milner system (Hindley, 1969; Milner, 1978; Damas and Milner, 1982). We elaborate on them in Sections 5.2 and 5.3, respectively.

5.1.2 Non-parametric Polymorphism

Non-parametric polymorphism refers to the ability to write functions which can be applied to arguments of different types. Yet, a non-parametric function can behave differently, depending on the argument types, as opposed to a parametric function which would behave uniformly.

There are several kinds of non-parametric polymorphism but, for the purposes of this thesis, we focus only on *ad-hoc polymorphism*. Ad-hoc polymorphism covers all cases where case analysis on types can be resolved at compile-time, as opposed to *type dispatch*, which describes a run-time dependency on case analysis on types.

Ad-hoc Polymorphism

One prime example of ad-hoc polymorphism put to good use is Java's overloaded operators. Consider for example operator "+", denoting addition, as used in the following example:

```
public class OverloadingExample {
  public static void main(String[] args) {
    int x = 1 + 2;
    System.out.println(x);

    String y = "Hello " + "World!";
    System.out.println(y);
  }
}
```

In the above program, operator "+" is used to denote two different operations: for the computation of `x`, integer addition is invoked, while for the computation of `y`, "+" performs string concatenation. Indeed, running the program gives:

```
$ java OverloadingExample
3
Hello World!
```

However, operator overloading in Java has several limitations:

1. It is closed. The language includes a predefined set of overloaded operators but the programmer cannot define more overloaded operators or extend existing ones to operate on new types.
2. Overloading does not propagate. While an operator can be used on terms of multiple types, its invocation *must* choose a type. In short, the enclosing function cannot be overloaded.

In Haskell, ad-hoc polymorphism (also known as overloading) is supported by means of type classes (Wadler and Blott, 1989). Type classes in Haskell are much more flexible than overloaded Java operators, but we defer this discussion to Section 5.4 and Chapter 6, where we discuss type classes at length.

5.2 System F: The Polymorphic Lambda Calculus

System F is one of the most important calculi in the history of functional programming languages, due to its expressivity/complexity ratio. It has been developed independently by Girard (1972) and Reynolds (1974, 1983), hence it is also known as the Girard/Reynolds system.[1]

Due to its expressive power, it is often used as an intermediate language for compiling functional languages (e.g., by ML and Haskell), after it has been extended with algebraic data types. In this section we discuss all formal aspects of this calculus; extensions of the calculus are considered later in this thesis (see Sections 6.1 and 8.4).

5.2.1 Calculus Specification

Figure 5.1 presents the syntax, typing, and operational semantics for System F. We discuss each in more detail below.

Syntax The syntax of System F comprises two sorts: types υ and terms t. A type υ can either be a type variable (a), an arrow type ($\upsilon_1 \rightarrow \upsilon_2$), or a type abstraction ($\forall a.\ \upsilon$). Terms comprise a λ-calculus, extended with type abstraction ($\Lambda a.\ t$) and type application ($t\ \upsilon$).

Typing The typing specification consists of two relations, each concerned with one of the syntactic sorts. Both relations make use of typing environments Γ, which keep track of bound type variables and bound term variables, along with their type (the syntax of typing environments is also given in Figure 5.1).

Well-formedness of types takes the form $\Gamma \vdash_{\mathtt{TY}} \upsilon$ and specifies when a type υ is well-formed under a typing environment Γ. Since System F is *uni-kinded* (i.e., types are not classified into categories; all of them have kind $\star$), the relation essentially captures the *well-scopedness* of a type under a given environment.

Term typing is also presented in Figure 5.1 and takes the form $\Gamma \vdash_{\mathtt{TM}} t : \upsilon$. The relation specifies when a term t has type υ, under an environment Γ. The

[1]It is also known as the polymorphic λ-calculus and the second-order calculus.

Figure 5.1 System F Specification: Syntax, Typing, and Operational Semantics

$$
\begin{array}{lcll}
\upsilon & ::= & a \mid \upsilon_1 \to \upsilon_2 \mid \forall a.\ \upsilon & \textit{type} \\
t & ::= & x \mid \Lambda a.\ t \mid t\ \upsilon \mid \lambda(x : \upsilon).\ t \mid t_1\ t_2 & \textit{term} \\
\\
\Gamma & ::= & \bullet \mid \Gamma, a \mid \Gamma, x : \upsilon & \textit{typing environment}
\end{array}
$$

$\boxed{\Gamma \vdash_{\mathtt{TY}} \upsilon}$ Type Well-formedness

$$
\frac{a \in \Gamma}{\Gamma \vdash_{\mathtt{TY}} a}
\qquad
\frac{\Gamma \vdash_{\mathtt{TY}} \upsilon_1 \qquad \Gamma \vdash_{\mathtt{TY}} \upsilon_2}{\Gamma \vdash_{\mathtt{TY}} \upsilon_1 \to \upsilon_2}
\qquad
\frac{a \notin \mathit{fv}(\Gamma) \qquad \Gamma, a \vdash_{\mathtt{TY}} \upsilon}{\Gamma \vdash_{\mathtt{TY}} \forall a.\ \upsilon}
$$

$\boxed{\Gamma \vdash_{\mathtt{TM}} t : \upsilon}$ Term Typing

$$
\frac{(x : \upsilon) \in \Gamma}{\Gamma \vdash_{\mathtt{TM}} x : \upsilon}
\qquad
\frac{a \notin \Gamma \qquad \Gamma, a \vdash_{\mathtt{TM}} t : \upsilon}{\Gamma \vdash_{\mathtt{TM}} \Lambda a.\ t : \forall a.\ \upsilon}
\qquad
\frac{\Gamma \vdash_{\mathtt{TM}} t : \forall a.\ \upsilon \qquad \Gamma \vdash_{\mathtt{TY}} \upsilon_1}{\Gamma \vdash_{\mathtt{TM}} t\ \upsilon_1 : [\upsilon_1/a]\upsilon}
$$

$$
\frac{\Gamma \vdash_{\mathtt{TM}} t_1 : \upsilon_2 \to \upsilon_1 \qquad \Gamma \vdash_{\mathtt{TM}} t_2 : \upsilon_2}{\Gamma \vdash_{\mathtt{TM}} t_1\ t_2 : \upsilon_1}
\qquad
\frac{\begin{array}{c} x \notin \mathit{dom}(\Gamma) \\ \Gamma \vdash_{\mathtt{TY}} \upsilon_1 \qquad \Gamma, x : \upsilon_1 \vdash_{\mathtt{TM}} t : \upsilon_2 \end{array}}{\Gamma \vdash_{\mathtt{TM}} \lambda(x : \upsilon_1).\ t : \upsilon_1 \to \upsilon_2}
$$

$\boxed{t_1 \longrightarrow t_2}$ Small-step Call-by-name Operational Semantics

$$
\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}
\qquad
\frac{}{(\Lambda a.\ t)\ \upsilon \longrightarrow [\upsilon/a]t}
\qquad
\frac{}{(\lambda(x : \upsilon).\ t_1)\ t_2 \longrightarrow [t_2/x]t_1}
$$

relation is syntax-directed on the shape of the term, hence we have one rule for each syntactic form. This is significant, since it means that the specification of typing for System F can also be used as a type checking algorithm. Essentially, each term encodes its own typing derivation.

Operational Semantics Finally, Figure 5.1 presents the small-step, call-by-name operational semantics for System F. The relation takes the form $t \longrightarrow t'$, meaning that t can evaluate to t' in one step.

The first rule is a *congruence* rule, allowing the reduction of the function in a term application. Since we give call-by-name semantics, we never need to evaluate the argument. The next two rules handle β*-redexes*, for type and term application, respectively.

5.2.2 Meta-theoretical Properties

System F has several interesting meta-theoretical properties which we now present.[2] These properties are of importance for all calculi, so we use System F as an opportunity to introduce them; the forthcoming chapters refer to and revise these definitions accordingly.

Type Safety The most important property of a calculus' type system is that of *type safety*. Alternatively, as stated by Milner (1978):

Well-typed programs cannot "go wrong".

The standard technique for proving type safety is due to Wright and Felleisen (1994), by means of proving first two other properties: *preservation* (also known as *subject reduction*) and *progress*.

Preservation Evaluation of a well-typed term does not affect its typeability; reducing a well-typed term always results in a well-typed term.

Progress Evaluation of well-typed terms should never be "stuck"; they should either reduce to a *value* or be able to reduce further.

By values we mean terms that cannot evaluate any further; for System F this amounts to term and type abstractions:

$$V \quad ::= \quad \lambda(x : \upsilon).\ t \mid \Lambda a.\ t \qquad\qquad \textit{values}$$

Using the relations of typing and operational semantics, we can formalize the aforementioned properties (preservation and progress) for System F as follows:

Theorem 3 (Preservation)**.** *If* $\Gamma \vdash_{\mathrm{TM}} t : \upsilon$ *and* $t \longrightarrow t'$ *then* $\Gamma \vdash_{\mathrm{TM}} t' : \upsilon$.

Theorem 4 (Progress)**.** *If* $\Gamma \vdash_{\mathrm{TM}} t : \upsilon$ *then* t *is a value or* $\exists t'.\ t \longrightarrow t'$.

Notice that System F (and several other calculi) supports a stronger notion of preservation than the one we informally stated above. Well-typed System F terms not only reduce to well-typed terms, but also they preserve their type.

[2] A more detailed exposition of these properties and their proofs can be found in the work of Pierce (2002, Chapter 23).

Strong Normalization *Strong normalization* is another important property of calculi. A calculus is said to possess the strong normalization property if *all* well-typed programs expressed in the calculus halt after a finite number of evaluation steps. Accordingly, a (well-typed) term is said to be *normalizable* if its evaluation terminates. System F has the strong normalization property:

Theorem 5 (Strong Normalization)**.** *If* $\Gamma \vdash_{\text{TM}} t : \upsilon$ *then* $\exists n.\ t \longrightarrow^{n} V$*.*

where $t \longrightarrow^{n} t'$ can be read as *"t evaluates to t' in n steps".* In general, full-blown programming languages do not (usually) posses the strong normalization property. Indeed, the extensions of System F we will consider later in this thesis (Sections 6.1 and 8.4) do not. Nevertheless, type safety is a desirable property that can usually hold independently of strong normalization.

Type Erasure As illustrated in the operational semantics of System F (Figure 5.1), type annotations do not affect the operational behavior of a term; relation $t \longrightarrow t'$ ignores type annotations on λ-bound variables. In short, if a program is well-typed, type information plays no role in its run-time behavior; it can be omitted. This property is widely known as *type erasure*, and is a property that System F possesses.

For System F terms specifically, we can define an erasure function as follows:

Definition 2 (Erasure)**.** *The erasure of a System F term t is defined as follows:*

$$\begin{array}{lcl}
\mathit{erase}(x) & = & x \\
\mathit{erase}(\Lambda a.\ t) & = & \lambda\_.\ \mathit{erase}(t) \\
\mathit{erase}(t\ \upsilon) & = & \mathit{erase}(t)\ \bullet \\
\mathit{erase}(\lambda(x:\upsilon).\ t) & = & \lambda x.\ \mathit{erase}(t) \\
\mathit{erase}(t_1\ t_2) & = & \mathit{erase}(t_1)\ \mathit{erase}(t_2)
\end{array}$$

In short, type annotations are erased, type abstraction is translated into term abstraction (where the bound variable is irrelevant; hence the _), and type application is translated into a term application, to a "dummy" value "•" (one could also use unit `()`). Term variables x and term applications $(t_1\ t_2)$ remain unchanged.

Theorem 6 (Type Erasure)**.** *If* $\mathit{erase}(t) = t_e$*, then either (a) both t and t_e are normal forms according to their respective evaluation relations (where t_e is a term of the untyped λ-calculus (Church, 1936)), or (b)* $t \longrightarrow t'$ *and* $t_e \longrightarrow_e t'_e$ *and* $\mathit{erase}(t') = t'_e$*.*

Though not very important for theoretical purposes, type erasure is a very desirable property in practice. If types do not affect the run-time behavior of a program, the ability to erase them means that the additional overhead can be avoided. Hence, most compilers for full-blown programming languages use type information during compilation but omit it during code generation (when possible). Indeed, the current intermediate language used by GHC (which we elaborate on in Section 8.4) supports type-erasure, despite the multitude of complex features it supports. As we illustrate later in this thesis though, the distinction between erasable and non-erasable objects significantly complicates the development of functional dependencies (see Chapter 8).

5.3 Type Reconstruction and Elaboration

System F is a powerful calculus but requires type abstractions and applications to be explicit, as well as explicit type annotations on all λ-abstractions. This can easily become cumbersome, so most programming languages allow the omission of type annotations in source programs. It then becomes the compiler's task to reconstruct all missing types from the program text. Such a procedure is known as *type reconstruction* or *type inference*.

Additionally, once the missing types are retrieved by the compiler, it is often useful to preserve them through several compilation phases (e.g., for applying type-preserving optimizations). To this end, most compilers use intermediate languages for the internal representation of programs, which are in principle more explicit than the source language. The process of translating source programs into the intermediate representation is often referred to as *elaboration*.

In this section we thoroughly discuss the notions of type inference and elaboration, which will be heavily referred to in the remainder of this thesis. Both procedures are presented using the Hindley-Damas-Milner system (HM) (Hindley, 1969; Milner, 1978; Damas and Milner, 1982) as our source language, and System F as the language we elaborate HM programs to.

5.3.1 The Challenge of Type Reconstruction

The question of whether all or most type annotations can be erased from the source program and be reconstructed has several aspects. Firstly, we need to know whether type reconstruction is possible. Secondly, if it is, the next question is how it can be achieved. Thirdly, we want a type inference algorithm to be *well-behaved*, which involves several required properties it should possess.

Unfortunately, Wells (1993) has proven that type reconstruction is undecidable for System F in the general case. The main problem is rooted in the lack of the *principal type property* for the erased system.[3]

Principal Type Property We say that a type system has the principal type property if every term that is typeable with respect to the type system has a *principal type.* By principal type, we mean the most general type; all other types the term can have are instances of the principal type.

As an example, consider function *fst*, which extracts the first element of a tuple:

$$\mathit{fst}\ (x, y) = x$$

What type should be assigned to *fst*? All of the following are possible:

$$\begin{array}{ll} \mathit{fst} :: (\mathit{Int}, \mathit{Char}) \to \mathit{Int} & \mathit{fst} :: \forall d.\ (\mathit{Int}, d) \to \mathit{Int} \\ \mathit{fst} :: \forall c.\ (c, \mathit{Int}) \to c & \mathit{fst} :: \forall a.\ \forall b.\ (a, b) \to a \end{array}$$

It is not difficult to see that the first three signatures are more specific than the last, which is indeed the principal type; all three can be obtained by appropriately instantiating the principal type ($\forall a.\ \forall b.\ (a, b) \to a$).

The notion of instantiation of the principal type indicates the existence of a *subsumption* relation (partial order) between types: two types can either be incomparable, or one is more general than the other. We formalize this notion in Section 5.3.3.

Notice that the principal type property is not a property of a type inference algorithm but a property of the type system itself. Of course, if a type system has the principal type property, it is desirable that a type inference algorithm can infer the principal type. We return to this in Section 5.3.3.

System F without type annotations does not possess the principal type property, due to *higher-rank types* and *impredicativity.*

Higher-rank Types We say that a function has a higher-rank type if one of its arguments has a polymorphic type. To illustrate how this poses an issue for type inference, consider the following example from Vytiniotis et al. (2008):

$$f\ \ \mathit{get} = (\mathit{get}\ 3, \mathit{get}\ \ \mathit{True})$$

[3]That is, the untyped λ-calculus (Church, 1936) with the type system of System F.

What type should be inferred for f? Both of the following are valid, yet *incomparable* (none is more general than the other):

$$\begin{aligned} &f :: (\forall a.\ a \to a) \to (\mathit{Int}, \mathit{Bool}) \\ &f :: (\forall a.\ a \to \mathit{Int}) \to (\mathit{Int}, \mathit{Int}) \end{aligned}$$

In fact, f does not have a principal type, i.e., there exists no type that is more general than the two above.

Impredicativity (First-class Polymorphism) A type system is called *impredicative* if it allows instantiation of type variables with polymorphic types. Another succinct example from Vytiniotis et al. (2008) illustrates how impredicativity can render type inference problematic:

$$\begin{array}{lll} \mathit{choose} & :: & \forall a.\ a \to a \to a \\ \mathit{id} & :: & \forall b.\ b \to b \end{array}$$

What type should be inferred for the expression ($\mathit{choose}\ \mathit{id}$)? Again, we have (at least) two options:

$$\begin{aligned} &\mathit{choose}\ \mathit{id} :: \forall a.\ (a \to a) \to (a \to a) \\ &\mathit{choose}\ \mathit{id} :: (\forall b.\ b \to b) \to (\forall b.\ b \to b) \end{aligned}$$

The above types are again incomparable (their shape differs); the expression once again has no principal type.

Summary In summary, higher-rank types and impredicative polymorphism pose significant difficulties for annotation-free type inference. Thus, most functional programming languages implement more restrictive type systems. The most notable (and well-behaved) calculus is the one designed by Hindley, Damas, and Milner (Hindley, 1969; Milner, 1978; Damas and Milner, 1982), which is the focus of the remainder of this section.

5.3.2 The Hindley-Damas-Milner System

In order to avoid the aforementioned problems, the Hindley-Damas-Milner (HM) system internalizes the restrictions of predicativity and rank-1 polymorphism, by stratifying types into two categories: *monomorphic types* (also known as *monotypes*), and type schemas (also known as *polytypes*). In the remainder of this section we discuss the formal aspects of the Hindley-Damas-Milner system.

Figure 5.2 Hindley-Damas-Milner: Syntax

$$\begin{array}{rcll} \tau & ::= & a \mid \tau_1 \rightarrow \tau_2 & \textit{monotype} \\ \sigma & ::= & \tau \mid \forall a.\ \sigma & \textit{type scheme} \\ e & ::= & x \mid \lambda x.\ e \mid e_1\ e_2 & \textit{term} \end{array}$$

Syntax

The syntax of types and terms for the HM system is presented in Figure 5.2. A monotype is denoted by τ and can be either a type variable a or a function type $\tau_1 \rightarrow \tau_2$. A polytype σ can consist of any number of quantifiers, followed by a monotype. Hence, a polytype σ can always be written in the form $\forall \overline{a}.\ \tau$, which is known as *prenex normal form* (Hilbert and Bernays, 1934).

Terms e comprise a λ-calculus: they consist of term variables x, lambda abstractions $\lambda x.\ e$, and term applications $e_1\ e_2$.

Specification of Typing and Elaboration into System F

Figure 5.3 presents the specification of typing and elaboration into System F for HM types and terms.

Type Well-formedness Relation $\Gamma \vdash_{\text{TY}} \sigma \rightsquigarrow \upsilon$ captures type well-formedness and can be read as *"under typing environment* Γ*, type* σ *is well-formed and can be elaborated into System F type* υ*"*.

Similarly to System F, HM is *uni-kinded* so the relation essentially checks that type σ is well-scoped under environment Γ. Furthermore, due to polytypes σ being a strict subset of System F types υ, elaboration performs the identity transformation.

Term Typing Term typing takes the form $\Gamma \vdash_{\text{TM}} e : \sigma \rightsquigarrow t$ and is also presented in Figure 5.3. It checks that, under a typing environment Γ, a term e has type σ and elaborates to a System F term t.

In contrast to System F, the rules are not syntax-directed; since typing information is implicit, a term can be assigned multiple types. Rule TmVar handles term variables and is straightforward. Rules $(\forall I)$ and $(\forall E)$ handle type abstraction and type application, respectively. Notice how Rule $(\forall E)$ uses

Figure 5.3 Hindley-Damas-Milner: Typing and Elaboration Specification

$$\text{Typing Environment} \quad \Gamma ::= \bullet \mid \Gamma, a \mid \Gamma, x : \sigma$$

$\boxed{\Gamma \vdash_{\texttt{TY}} \sigma \rightsquigarrow \upsilon}$ HM Type Well-formedness with Elaboration into System F

$$\frac{a \in \Gamma}{\Gamma \vdash_{\texttt{TY}} a \rightsquigarrow a}\ \textsc{TyVar} \qquad \frac{\Gamma \vdash_{\texttt{TY}} \tau_1 \rightsquigarrow \upsilon_1 \qquad \Gamma \vdash_{\texttt{TY}} \tau_2 \rightsquigarrow \upsilon_2}{\Gamma \vdash_{\texttt{TY}} \tau_1 \to \tau_2 \rightsquigarrow \upsilon_1 \to \upsilon_2}\ \textsc{TyArr}$$

$$\frac{a \notin \Gamma \qquad \Gamma, a \vdash_{\texttt{TY}} \sigma \rightsquigarrow \upsilon}{\Gamma \vdash_{\texttt{TY}} \forall a.\ \sigma \rightsquigarrow \forall a.\ \upsilon}\ \textsc{TyAll}$$

$\boxed{\Gamma \vdash_{\texttt{TM}} e : \sigma \rightsquigarrow t}$ HM Term Typing with Elaboration into System F

$$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash_{\texttt{TM}} x : \sigma \rightsquigarrow x}\ \textsc{TmVar} \qquad \frac{a \notin \Gamma \qquad \Gamma, a \vdash_{\texttt{TM}} e : \sigma \rightsquigarrow t}{\Gamma \vdash_{\texttt{TM}} e : \forall a.\ \sigma \rightsquigarrow \forall a.\ t}\ (\forall I)$$

$$\frac{\Gamma \vdash_{\texttt{TM}} e : \forall a.\ \sigma \rightsquigarrow t \qquad \Gamma \vdash_{\texttt{TY}} \tau \rightsquigarrow \upsilon}{\Gamma \vdash_{\texttt{TM}} e : [\tau/a]\sigma \rightsquigarrow t\ \upsilon}\ (\forall E)$$

$$\frac{x \notin dom(\Gamma) \qquad \Gamma \vdash_{\texttt{TY}} \tau_1 \rightsquigarrow \upsilon_1 \qquad \Gamma, x : \tau_1 \vdash_{\texttt{TM}} e : \tau_2 \rightsquigarrow t}{\Gamma \vdash_{\texttt{TM}} \lambda x.\ e : \tau_1 \to \tau_2 \rightsquigarrow \lambda(x : \upsilon_1).\ t_2}\ (\to I)$$

$$\frac{\Gamma \vdash_{\texttt{TM}} e_1 : \tau_1 \to \tau_2 \rightsquigarrow t_1 \qquad \Gamma \vdash_{\texttt{TM}} e_2 : \tau_1 \rightsquigarrow t_2}{\Gamma \vdash_{\texttt{TM}} e_1\ e_2 : \tau_2 \rightsquigarrow t_1\ t_2}\ (\to E)$$

meta-variable τ to enforce the predicativity of the system: a polymorphic type can only be instantiated with a monotype. Also, the elimination of the quantifier in e's type is reflected in the elaborated term in the form of an explicit type application.

Similarly, Rules $(\to I)$ and $(\to E)$ capture term abstraction and application, respectively. Both rules are mostly straightforward. The most interesting aspect of them lies in the usage of meta-variables. In Rule $(\to I)$, a λ-bound variable can only have a monotype, effectively enforcing the rank-1 restriction. Similarly, Rule $(\to E)$ enforces the argument to have a monomorphic type.

Type Reconstruction and Elaboration into System F

We now turn to the aspect of type inference and elaboration into System F for the system of Figure 5.3.

Algorithm Structure The algorithm proceeds in three steps:

Constraint Generation In the first phase, a type is inferred for the given term, using fresh type variables for yet unknown types. Constraints are also gathered, capturing restrictions on type variables imposed by the program structure.

Constraint Solving In the second phase, the constraints are solved, giving rise to a type substitution which refines the unknown type variables with concrete types.

Type Refinement and Generalization In the third phase, the substitution is applied to the inferred type (and elaborated term) which is then *generalized*: all non-unified type variables are universally quantified in the result.

All three parts of the algorithm are formally presented in Figure 5.4. We elaborate on each below. Before we do so, we introduce two intermediate constructs: type substitutions θ and sets of equality constraints E:

$$\begin{array}{rcll} \theta & ::= & \bullet \mid \theta \cdot [\tau/a] & \textit{type substitution} \\ E & ::= & \bullet \mid E, \tau_1 \sim \tau_2 & \textit{type equalities} \end{array}$$

A type substitution θ maps type variables to monotypes, facilitating the predicativity restriction. Sets of equalities E include equalities between monotypes. We represent sets of equalities as snoc-lists but this is mere notation; the order of equalities is irrelevant.

Constraint Generation Constraint generation, the first part of the algorithm, is performed by relation $\Gamma \vdash_{\text{TM}} e : \tau \rightsquigarrow t \mid E$. Given a typing environment Γ and a term e, it infers a monotype τ, gives rise to a set of *wanted* equality constraints E, and elaborates e into a System F term t. Each rule handles a different syntactic form:

Rule TmVar handles term variables. Polymorphic types $\forall \overline{a}.\ \tau$ are instantiated with fresh *unification*[4] variables $\overline{b}$. This is also reflected in the elaborated

[4]That is, type variables representing yet unknown types. During constraint solving such type variables are replaced by actual types.

Figure 5.4 Hindley-Damas-Milner: Type Inference with Elaboration

$\boxed{\Gamma \vdash_{\texttt{TM}} e : \tau \rightsquigarrow t \mid E}$ Constraint Generation with Elaboration

$$\frac{(x : \forall \bar{a}.\ \tau) \in \Gamma \qquad \theta = [\bar{b}/\bar{a}] \qquad \bar{b} \text{ fresh}}{\Gamma \vdash_{\texttt{TM}} x : \theta(\tau) \rightsquigarrow x\ \bar{b} \mid \bullet}\ \textsc{TmVar}$$

$$\frac{x \notin dom(\Gamma) \qquad a \text{ fresh} \qquad \Gamma, x : a \vdash_{\texttt{TM}} e : \tau \rightsquigarrow t \mid E}{\Gamma \vdash_{\texttt{TM}} \lambda x.\ e : a \to \tau \rightsquigarrow \lambda(x : a).\ t \mid E}\ (\to I)$$

$$\frac{\Gamma \vdash_{\texttt{TM}} e_1 : \tau_1 \rightsquigarrow t_1 \mid E_1 \qquad \Gamma \vdash_{\texttt{TM}} e_2 : \tau_2 \rightsquigarrow t_2 \mid E_2 \qquad a \text{ fresh}}{\Gamma \vdash_{\texttt{TM}} e_1\ e_2 : a \rightsquigarrow t_1\ t_2 \mid E_1, E_2, \tau_1 \sim \tau_2 \to a}\ (\to E)$$

$\boxed{unify(E) = \theta_\perp}$ Type Unification Algorithm

$$\begin{array}{lcll}
unify(\bullet) & = & \bullet & \\
unify(E, b \sim b) & = & unify(E) & \\
unify(E, b \sim \tau) & = & unify(\theta(E)) \cdot \theta & b \notin fv(\tau) \wedge \theta = [\tau/b] \\
unify(E, \tau \sim b) & = & unify(\theta(E)) \cdot \theta & b \notin fv(\tau) \wedge \theta = [\tau/b] \\
unify(E, (\tau_1 \to \tau_2) \sim (\tau_3 \to \tau_4)) & = & unify(E, \tau_1 \sim \tau_3, \tau_2 \sim \tau_4) &
\end{array}$$

$\boxed{\vdash_{\texttt{GEN}} e : \sigma \rightsquigarrow t}$ Type Refinement and Generalization

$$\frac{\bullet \vdash_{\texttt{TM}} e : \tau \rightsquigarrow t \mid E \qquad unify(E) = \theta \qquad \bar{a} = fv(\theta(\tau))}{\vdash_{\texttt{GEN}} e : \forall \bar{a}.\ \theta(\tau) \rightsquigarrow \Lambda \bar{a}.\ \theta(t)}\ \textsc{Gen}$$

term, where type instantiation becomes explicit via type application. Since the appearance of a term variable alone implies no additional constraints, the generated set of equalities is empty.

Rule $(\to I)$ handles term abstractions. First, we generate a fresh type variable to capture the unknown type of the λ-bound term variable. Then, the body of the λ-abstraction is recursively checked.

Finally, Rule $(\to E)$ handles term applications. After we infer a type for each subterm, we generate a fresh type variable a to capture the result type and record that the application is well-formed if the type of e_1 is a function type, and the argument type matches that of e_2.

Constraint Solving Constraint solving is implemented by function *unify* which takes a set of equalities E and produces as a result a type substitution θ. Unification is a partial function, which we capture in the signature: $\mathit{unify}(E) = \theta_{\perp}$. The original algorithm has been designed by Robinson (1965) but function *unify* implements a variant of a more efficient algorithm, originally designed by Martelli and Montanari (1982).

The first two clauses are straightforward: In case the set of equalities is empty, the result is the trivial (empty) substitution. In case both sides of the inspected equality are the same type variable, the equality adds no information and is thus dropped.

The third and fourth clauses are identical, since equality is symmetric. These cases handle equalities whose one side is occupied by a type variable. Before we unify b with τ, we need to ensure that b does not appear in the free variables of τ, a check known in the literature as the *occurs check*. Unification of a type variable a with a type containing a would lead to a being unified with an infinite type. For example, the solution of equation $a \sim [a]$ would be $a = [[[\ldots]]]$.

Finally, the fifth clause handles cases where both sides of the equality are function types. The clause decomposes the equality to two *simpler* ones (the types involved are smaller), using the knowledge that the function type constructor is *injective*,[5] and recurses.

In all other cases, unification fails.

Type Refinement and Generalization The above two procedures (constraint generation and solving) are invoked by Rule GEN. For a given term e, we first infer a monotype τ and a set of wanted equality constraints E. Then, we invoke the unification algorithm, resulting in a solution θ. If unification is successful, the resulting substitution θ is used to refine the unification variables in τ, as well as the elaborated term t.

Unification variables that remain free after applying the substitution can be handled in two different ways:

Specialization In a closed program setting, free variables after type inference represent terms that are not used by the program. Hence, their type is irrelevant and can be instantiated freely. The approach taken in these

[5]That is, $\tau_1 \to \tau_2 \sim \tau_3 \to \tau_4$ implies $\tau_1 \sim \tau_3$ and $\tau_2 \sim \tau_4$.

cases is to instantiate (specialize) free variables with a trivial type (e.g. `()`).[6]

Generalization All type variables that are not unified with concrete types can be abstracted over; the term under consideration behaves parametrically over them. This is the approach that HM takes (as well as most functional languages).

Standard HM takes the second approach: in Rule Gen we collect all free variables in the substituted type and abstract over them in the resulting type $\forall \overline{a}.\ \theta(\tau)$. In the elaborated term, this becomes syntactically explicit via type abstractions: $\Lambda \overline{a}.\ \theta(t)$. Since HM types (both monotypes and polytypes) are a strict subset of System F types, we simplify matters by applying a source-level substitution θ directly on a System F term. In cases where source and target types differ, one would have to elaborate the substitution to refer to target types before applying it to System F terms. This is the case for more advanced systems, like the ones we present in the remainder of Part II.

Example To see how all the above come together, consider performing type inference with elaboration on the following expression:

$$\lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x)$$

First, constraint generation assigns the following types

$$f : a \qquad g : b \qquad x : c \qquad (g\ x) : d \qquad f\ (g\ x) : e$$

where a, b, c, d, and e are fresh type variables. Due to the two applications ($(g\ x)$ and $(f\ (g\ x))$), the following equalities are also generated (via Rule $(\rightarrow E)$):

$$E = \{\ b \sim c \rightarrow d, a \sim d \rightarrow e\ \}$$

The generated System F term simply annotates the λ-bound variables with their types:

$$\lambda(f : a).\ \lambda(g : b).\ \lambda(x : c).\ f\ (g\ x)$$

Second, constraint solving (unification) successfully replaces equalities E with type substitution $\theta = [(d \rightarrow e)/a, (c \rightarrow d)/b]$. Hence, the types of all subterms are refined to

$$f : d \rightarrow e \qquad g : c \rightarrow d \qquad x : c \qquad (g\ x) : d \qquad f\ (g\ x) : e$$

[6]See also the work of Bjørner (1994), which introduces algorithm M. Just as algorithm W computes *most general typing derivations*, Algorithm M computes *least typing derivations*, by specializing free variables with concrete types.

and the elaborated term to

$$\lambda(f : d \to e).\ \lambda(g : c \to d).\ \lambda(x : c).\ f\ (g\ x)$$

Finally, generalization abstracts over the remaining type variables, to compute the overall type

$$\forall d.\ \forall e.\ \forall c.\ (d \to e) \to (c \to d) \to c \to e$$

which is also reflected in the elaborated System F term

$$\Lambda d.\ \Lambda e.\ \Lambda c.\ \lambda(f : d \to e).\ \lambda(g : c \to d).\ \lambda(x : c).\ f\ (g\ x)$$

5.3.3 Meta-theoretical Properties

The HM system has several meta-theoretical properties, which we now discuss.[7] In cases where we want to refer to the typing relation without the elaboration, we simply omit the elaboration-related aspects of the relations of Figure 5.3. That is, we write $\Gamma \vdash_{\mathtt{TM}} e : \sigma$ instead of $\Gamma \vdash_{\mathtt{TM}} e : \sigma \rightsquigarrow t$, and similarly for the other relations.

Termination of Type Inference

First, type inference for the HM system is decidable and terminating:

Theorem 7 (Termination)**.** *The type inference algorithm of Figure 5.4 terminates on all inputs.*

Principality of Types

Next, the specification of HM typing (Figure 5.3) possesses the principal type property:

Theorem 8 (Principal Types)**.** *If e is well-typed, then there exists a type σ_0 (the principal type), such that $\Gamma \vdash_{\mathtt{TM}} e : \sigma_0$ and, for all σ such that $\Gamma \vdash_{\mathtt{TM}} e : \sigma$, we have that $\Gamma \models \sigma_0 \preceq \sigma$.*

Here relation $\Gamma \models \sigma_0 \preceq \sigma$ defines *type subsumption*:

$$\frac{\Gamma, \overline{a} \vdash_{\mathtt{TY}} \tau_i \qquad \tau_a = [\overline{\tau}/\overline{b}]\tau_b}{\Gamma \models (\forall \overline{a}.\ \tau_a) \preceq (\forall \overline{b}.\ \tau_b)}\ (\preceq)$$

[7] A more detailed exposition of these properties and their proofs can be found in the work of Pierce (2002, Chapter 22).

In this simple setting, type subsumption accounts for instantiation and reordering of universal quantifiers. In later chapters we consider more complicated specifications of type subsumption.

Furthermore, the inference algorithm of Figure 5.4 infers the principal type:

Theorem 9 (The Algorithm Infers Principal Types). *If* $\vdash_{\mathtt{GEN}} e : \sigma$*, then for all* σ' *such that* $\bullet \vdash_{\mathtt{TM}} e : \sigma'$*, we have that* $\bullet \models \sigma \preceq \sigma'$*.*

Notice that type subsumption has a very direct interpretation in the elaboration of a term: the first premise of Rule $(\preceq)$ $(\Gamma, \overline{a} \vdash_{\mathtt{TY}} \overline{\tau})$ captures type abstraction (over variables $\overline{a}$) and the second premise captures type application (on types $\overline{\tau}$). That is, if $\Gamma \vdash_{\mathtt{TM}} e : \sigma_1 \rightsquigarrow t_1$, $\Gamma \vdash_{\mathtt{TM}} e : \sigma_2 \rightsquigarrow t_2$, and $\Gamma \models \sigma_1 \preceq \sigma_2$, we have that

$$\Gamma \vdash_{\mathtt{TM}} (\Lambda \overline{a}.\ t_2\ \overline{\tau}) : \sigma_1$$

where $\overline{a}$ and $\overline{\tau}$ are the corresponding parameters from premise $\Gamma \models \sigma_1 \preceq \sigma_2$.

Preservation of Typing Under Elaboration

Another important property of HM is that elaboration preserves typeability. That is, well-typed HM terms are elaborated into well-typed System F terms:

Theorem 10 (Type-preserving Elaboration). *If* $\Gamma \vdash_{\mathtt{TM}} e : \sigma \rightsquigarrow t$ *then* $\Gamma \vdash_{\mathtt{TM}} t : \sigma$*.*

In fact, Theorem 10 above states a stronger property: not only are typeable terms elaborated into typeable terms but additionally the target type is the elaboration of the source type.[8]

Soundness

The algorithm of Figure 5.4 is sound (in terms of both typing and elaboration), with respect to the specification of Figure 5.3:

Theorem 11 (Algorithm Soundness). *If* $\vdash_{\mathtt{GEN}} e : \sigma \rightsquigarrow t$*, then* $\bullet \vdash_{\mathtt{TM}} e : \sigma \rightsquigarrow t$*.*

[8]Since HM types are a subset of System F types we do not elaborate them to avoid notational clutter but one would formally write the above theorem as *"If* $\Gamma \vdash_{\mathtt{TM}} e : \sigma \rightsquigarrow t$ *then* $\mathit{elab}_{\mathtt{TE}}(\Gamma) \vdash_{\mathtt{TM}} t : \mathit{elab}_{\mathtt{TY}}(\sigma)$*"*, where functions $\mathit{elab}_{\mathtt{TE}}(\cdot)$ and $\mathit{elab}_{\mathtt{TY}}(\cdot)$ elaborate HM typing environments and types, respectively.

Completeness

The algorithm of Figure 5.4 is complete, with respect to the specification of Figure 5.3. That is, for any term that is typeable under the typing specification, the type inference algorithm succeeds:

Theorem 12 (Completeness of Type Inference)**.** *If* $\bullet \vdash_{\mathtt{TM}} e : \sigma$*, then* $\vdash_{\mathtt{GEN}} e : \sigma'$*, for some* σ'*.*

In fact, as we stated in Theorem 9 above, σ' is not just *some* type; it is the principal type.

Coherence

Finally, the elaboration algorithm of Figure 5.4 is *coherent*: for any expression e, any two derivations result in System F terms with the same operational behavior. In layman's terms, the behavior of a program is completely specified by the source text; the inference algorithm makes no choices. There are several ways to formally state this property; we choose one of the most compact:

Theorem 13 (Elaboration is Coherent)**.** *If* $\vdash_{\mathtt{GEN}} e : \sigma_1 \rightsquigarrow t_1$ *and* $\vdash_{\mathtt{GEN}} e : \sigma_2 \rightsquigarrow t_2$*, then* $\mathit{erase}(t_1) = \mathit{erase}(t_2) = e$*.*

Note that by function $\mathit{erase}(\cdot)$ we denote a stronger erasing function than the one we defined in Definition 2; in the above theorem, we use $\mathit{erase}(e)$ to denote an erasing function that erases *all* type information and structure from e:

$$\begin{array}{lcl}
\mathit{erase}(x) & = & x \\
\mathit{erase}(\Lambda a.\ t) & = & \mathit{erase}(t) \\
\mathit{erase}(t\ \upsilon) & = & \mathit{erase}(t) \\
\mathit{erase}(\lambda(x:\upsilon).\ t) & = & \lambda x.\ \mathit{erase}(t) \\
\mathit{erase}(t_1\ t_2) & = & \mathit{erase}(t_1)\ \mathit{erase}(t_2)
\end{array}$$

The difference between $\mathit{erase}(e)$ presented in Definition 2 and the one we define here lies in the treatment of type abstraction and type application: the latter erases both. Though the two have different properties, for call-by-name semantics (that we are targeting in this thesis) they can be considered equivalent.

Summary In summary, HM is a well-behaved system with several important meta-theoretical properties. It forms the basis for ML-style type systems (including Haskell's) and type classes have also been developed as an extension of the HM system. All the aforementioned properties hold for HM with type classes (see Chapter 6) and it is desirable that they also hold for all the extensions we develop in the forthcoming chapters. Hence, each of Chapters 7, 8, and 9 carefully refines these properties to reflect differences in the corresponding extended systems.

5.4 Type Classes: Ad-hoc Polymorphism in Haskell

"Mathematics is the art of giving the same name to different things."

—Henri Poincaré

We now turn to the core of this chapter: *type classes*. Type classes were first introduced by Wadler and Blott (1989) as a principled way to support ad-hoc polymorphism in Haskell, and have become one of Haskell's cornerstone features. They support sound, decidable, and efficient HM-based type inference.

They first became highly successful in Haskell (Peyton Jones, 2003), were later adopted by other declarative languages like Mercury (Henderson et al., 1996) and Coq (The Coq development team, 2004), and finally influenced the design of similar features (e.g., concepts for C++ (Gregor et al., 2006) and traits for Rust (Shärli et al., 2002; Ducasse et al., 2006)).

Furthermore, over the years type classes in Haskell have been the subject of many language extensions that increase their expressive power and enable new applications. Examples of such extensions include multi-parameter type classes (Jones et al., 1997), functional dependencies (Jones, 2000), and associated types (Chakravarty et al., 2005a).

In this section we focus on the original design of type classes. More specifically, we (a) describe type classes from the user's point of view, (b) revisit the interpretation of type classes with respect to (first-order) predicate logic, as well as more constructive theories like System F, and (c) introduce the basic terminology. The tone of this section is strictly informal; we formalize all aspects of type classes in Chapter 6.

5.4.1 Type Classes, Informally

Given that type classes were designed to introduce overloading in Haskell, the main motivation for a programmer to use type classes is the encapsulation of common behavior within a single interface. We illustrate below each aspect of their usage through a series of examples in plain Haskell.

Class Declarations As an example, consider type class Eq, as defined in the standard `Prelude`:[9]

$$\begin{array}{l}\textbf{class } Eq\ a \textbf{ where}\\ \quad (\texttt{==}) :: a \rightarrow a \rightarrow Bool\end{array}$$

The above declaration defines a new type class, called Eq, which captures types whose elements can be compared for equality. It provides a method, (==), which can be used to compare two elements of type a for equality. Operator (==) is said to be *overloaded*, since its implementation may vary, depending on the type of arguments it is supplied with. Hence, its actual type is

$$(\texttt{==}) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

Equipped with such a type class, one can write function *elem*, which checks whether a value of type a is an element of a list of type $[a]$, for *any* type a, as long as it is an instance of the Eq class:

$$\begin{array}{lll}elem :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool & & \\ elem\ \ x\ \ [\,] & = & False\\ elem\ \ x\ \ (y : ys) & = & (x \texttt{ == } y)\ \texttt{||}\ (elem\ x\ ys)\end{array}$$

The type of *elem* is what is known as a *qualified type* (Jones, 1992), since it qualifies the simple type $(a \rightarrow [a] \rightarrow Bool)$ with the logical predicate $(Eq\ a)$.

Of course, using function *elem* above on concrete values requires us to provide instances for the Eq class for the involved types. We elaborate on this next.

Concrete Type Class Instances A type class instance axiomatically declares that a type is an instance of a particular class, by means of providing implementations for the class methods. For example, we can make $Bool$ an instance of the Eq class as follows:

$$\begin{array}{l}\textbf{instance } Eq\ Bool \textbf{ where}\\ \quad x\ \texttt{==}\ y\ =\ eqBool\ x\ y\end{array}$$

where

$$\begin{array}{llll}eqBool :: Bool \rightarrow Bool \rightarrow Bool & & & \\ eqBool\ \ True & True & = & True\\ eqBool\ \ False & False & = & True\\ eqBool\ \ \_ & \_ & = & False\end{array}$$

It is important to note here that the way a type instantiates a class is completely up to the user. For example, the following instance which always considers

[9]Type class Eq contains one more method in the standard prelude $((\texttt{/=}) :: a \rightarrow a \rightarrow Bool)$, for inequality checking but we omit it for brevity.

two values of type *Bool* to be equal would also be acceptable (though not very sensible):

```
instance Eq Bool where
  _ == _ = True
```

Generic Type Class Instances Haskell allows one to give instances not only for concrete types, such as *Bool*, but also for generic datatypes, such as $[a]$:

```
instance Eq a ⇒ Eq [a] where
  []       == []       = True
  (x : xs) == (y : ys) = (x == y) && (xs == ys)
  _        == _        = False
```

Such an instance can be read as *"for any type a, if a is an instance of Eq then* $[a]$ *is also an instance of Eq"*. In short, such an instance does not make $[a]$ an instance of *Eq*, but merely specifies how one can be derived, under the *assumption* that *Eq a* is available.

The constraints that are left of the "⇒" form the *instance context* (in this case the instance context contains only one constraint, *Eq a*), while *Eq* $[a]$ is referred to as the *instance head*.

The second clause of the implementation of function (==) above illustrates some of the flexibility of type classes. First, expression (x == y) compares elements of type a, showing that constraints from the instance context can be exploited in the implementation of the methods. Second, expression (*xs* == *ys*) compares elements of type $[a]$: method implementations can be recursive, and also utilize the class instance being defined.

Superclasses Another feature of type classes is that of *superclasses*. We say that a class is a superclass of another class if the latter always implies the former. For example, the standard definition of the *Ord* class (which encodes all types whose elements can be compared for ordering) looks as follows:

```
class Eq a ⇒ Ord a where
  (<) :: a → a → Bool
```

In layman's terms, no type can be an instance of the *Ord* class unless it is also an instance of the *Eq* class. In practice, this manifests itself in two ways:

1. When a programmer specifies that a type is an instance of a class, the compiler ensures that the corresponding superclass constraints are satisfied. If not, the program is rejected.

2. When a constraint of the form $\mathit{Ord}\ \tau$ is available in a part of a program, constraint $\mathit{Eq}\ \tau$ can also be automatically derived by the system, for any type τ. As an example, consider the following function which sorts a list and removes duplicate entries:

$$\mathit{ordNub} = \mathit{map}\ \mathit{head}\ .\ \mathit{group}\ .\ \mathit{sort}$$

where

$$\begin{array}{lll} \mathit{map}\ \mathit{head} & :: & [[a]] \to [a] \\ \mathit{group} & :: & \mathit{Eq}\ a \Rightarrow [a] \to [[a]] \\ \mathit{sort} & :: & \mathit{Ord}\ a \Rightarrow [a] \to [a] \end{array}$$

Since Eq is a superclass of Ord, ordNub can be given type

$$\mathit{ordNub} :: \mathit{Ord}\ a \Rightarrow [a] \to [a]$$

instead of the more verbose

$$\mathit{ordNub} :: (\mathit{Eq}\ a, \mathit{Ord}\ a) \Rightarrow [a] \to [a]$$

5.4.2 Logical Reading of Type Classes

At the term level, type classes are used to capture function overloading. At the type level though, classes represent predicates over types. This is the approach we take here. In fact, the connection of type classes with first-order predicate logic has been the main inspiration for our work, to be presented in the forthcoming chapters. For now, we informally discuss the interpretation of existing type class features into first-order predicate logic; the logical interpretation of our extensions will be discussed in the corresponding chapters.

First-order Logic in a Nutshell

Syntax Firstly, logical *terms* t can take one of two forms: either a term variable x, or a saturated function application $f_n(\overline{t}^n)$, where f_n denotes a function symbol of arity n. The syntax of logical terms is inductively defined as follows:

$$t ::= x \mid f_n(\overline{t}^n) \qquad\qquad \mathit{term}$$

Term variables x usually range over a *domain of discourse*; for our purposes, Haskell's types play the role of logical terms and the domain of discourse is monotypes τ.

Secondly, logical *formulas* ϕ represent logical statements and are inductively defined as follows:

$$\begin{array}{rll} \phi ::= & \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \phi_1 \leftrightarrow \phi_2 & \textit{formula} \\ \mid & \neg\phi \mid \forall x.\ \phi \mid \exists x.\ \phi \mid p_n(\overline{t}^n) \mid t_1 = t_2 & \end{array}$$

The first row presents forms built by using the *binary logical connectives*: conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$), and biconditional ($\leftrightarrow$). Furthermore, if ϕ is a formula, its negation $\neg\phi$ is also a valid formula. Forms $\forall x.\ \phi$ and $\exists x.\ \phi$ capture universal and existential quantification, respectively. Finally, a formula can be formed by using a predicate symbol $p_n(\overline{t}^n)$ (of arity n) over n terms, or the equality symbol "=" over two terms (denoting the statement that the two terms are equal).

Well-scoped formulas[10] are known as *sentences*. In this thesis we are only concerned with sentences. Function symbols (f_n) and predicate symbols (p_n) are known as the *non-logical* symbols of the logic; when giving semantics to the subset of logic at hand, one must give an *interpretation* for all non-logical symbols. Also, formulas of the form $p_n(\overline{t}^n)$ and $t_1 = t_2$ are called *atomic*; constraints in Haskell '98 (Peyton Jones, 2003) are always atomic, and particularly of the form $p_1(t)$. In the forthcoming chapters, we lift this restriction in several ways (see Chapters 7 and 8).

Semantics The standard semantics for first-order predicate logic have been given by Tarski (1933). In essence, the interpretation of the logic (a) assigns a denotation to all non-logical symbols (predicate and function symbols), and (b) specifies a domain of discourse for term variables, which determines the range of the quantifiers.

The interpretation of a function symbol f_n is an n-ary function (in the mathematical sense). Predicate symbols p_n are interpreted *extensionally*, that is, as a finite enumeration of n-ary tuples. Then, formula $p_n(t_1, \ldots, t_n)$ is true iff the tuple consisting of the interpretation of all terms $t_1, \ldots, t_n$ is an element of the interpretation of p_n.

Datatype Declarations

Since our domain of discourse is types, a data type declaration introduces a new function symbol. For example, type *Bool* can be viewed as a function symbol

[10] A formula is well-scoped if all variables appearing in the formula are explicitly bound by a quantifier. A variable appearing outside the scope of a corresponding quantifier is called *"unbound"* or *"free"*.

of arity 0:

$$\textbf{data}\ \mathit{Bool} = \mathit{False} \mid \mathit{True}$$

We also refer to function symbols of arity 0 as *type constants*. Similarly, the list datatype introduces another function symbol, of arity 1:

$$\textbf{data}\ [a] = [\,] \mid a : [a]$$

Simply stated, the arity of the type constructor is the arity of the function symbol it introduces. Furthermore, type constructors are *uninterpreted*: if one were to assign Tarski's semantics (set-theoretic) to Haskell's types, a type constructor would be interpreted as itself (much like a type constant), as opposed to proper type-level functions which would reduce to possibly a different type. We return to this distinction in Chapter 8.

Class Declarations

Type class declarations also introduce non-logical symbols. For example, the declaration for *Eq*

$$\textbf{class}\ \mathit{Eq}\ a$$

introduces a new predicate symbol, of arity 1. In general, single-parameter type classes introduce predicate symbols of arity 1, and multi-parameter type classes (Jones et al., 1997) introduce predicate symbols of arity equal to the number of their parameters.

In case a class declaration specifies superclass constraints, the class declaration also gives rise to *logical implications*. Take for example the *Ord* class declaration:

$$\textbf{class}\ \mathit{Eq}\ a \Rightarrow \mathit{Ord}\ a$$

Haskell implicitly quantifies over type variables; with explicit quantification, the above class declaration corresponds to the following logical implication:

$$\forall a.\ \mathit{Ord}\ a \rightarrow \mathit{Eq}\ a$$

Since the satisfiability of $\mathit{Eq}\ \tau$ is a prerequisite for making τ an instance of *Ord*, $\mathit{Ord}\ \tau$ always implies $\mathit{Eq}\ \tau$, for any type τ.

Instance Declarations

Concrete instance declarations (i.e., instance declarations for concrete types) can also be directly interpreted as logical formulas. More specifically, an instance declaration like

$$\textbf{instance}\ \mathit{Eq}\ \mathit{Bool}$$

introduces an *axiom*, stating that *Eq Bool* is *true*. Generic instance declarations, like the *Eq* instance for lists we presented earlier

$$\textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a]$$

correspond to logical implications. The implicitly-quantified type variables from the instance head and context are again explicitly universally quantified, to give rise to the following formula:

$$\forall a.\ \mathit{Eq}\ a \to \mathit{Eq}\ [a]$$

5.4.3 Constructive Interpretation of Type Classes

The interpretation of type classes in terms of first-order logic sufficiently explains their logical meaning but leaves several of their aspects unspecified. The logical interpretation neither specifies how method overloading should behave at run-time, nor how to translate this source-level feature to an intermediate language akin to System F.

Following the approach of Wadler and Blott (1989), we now present an interpretation of type classes in terms of a constructive logic, by means of elaboration. We keep the presentation strictly informal, by presenting the original[11] translation of type classes into plain Haskell '98 (Peyton Jones, 2003); the formal presentation of this translation into System F can be found in the next chapter.

Class Declarations Each class declaration gives rise to a datatype declaration, which is a term-level encoding of the witness that a class constraint is satisfied. Consider for example the class declaration for *Eq*:

$$\begin{array}{l}\textbf{class}\ \mathit{Eq}\ a\ \textbf{where} \\ \quad (\texttt{==}) :: a \to a \to \mathit{Bool}\end{array}$$

Such a declaration is elaborated into a data declaration as the following:

$$\textbf{data}\ \mathit{EqD}\ a = \mathit{EqDict}\ (a \to a \to \mathit{Bool})$$

In essence, a term of type $(\mathit{EqD}\ \tau)$ encodes a *proof* that type τ is an instance of the *Eq* class. Such terms are known in the literature as *"dictionaries"* (Wadler and Blott, 1989). Of course, for such a term to be created, one must supply an implementation of the method, of type $(a \to a \to \mathit{Bool})$. Notice that even

[11]Most of the examples in this section can be found in the work of Wadler and Blott (1989).

though each instance provides its own implementation for the method, the extraction of the method from a dictionary of the appropriate type can be implemented generically:

$$\begin{array}{l}(\texttt{==}) :: \mathit{EqD}\ a \rightarrow a \rightarrow a \rightarrow \mathit{Bool} \\ (\texttt{==})\ (\mathit{EqDict}\ \mathit{eq}) = \mathit{eq}\end{array}$$

In case there are superclasses involved, the translation is slightly different. Let us consider for example the translation of the *Ord* class declaration:

$$\begin{array}{l}\textbf{class}\ \mathit{Eq}\ a \Rightarrow \mathit{Ord}\ a\ \textbf{where} \\ \quad (\texttt{<}) :: a \rightarrow a \rightarrow \mathit{Bool}\end{array}$$

As we mentioned earlier in Section 5.4.1, *Eq* being a superclass of *Ord* means that giving an instance for *Ord* τ requires that *Eq* τ is also proven. That is, while checking the instance declaration for *Ord* τ, the compiler needs to *create* a dictionary of type *EqD* τ. In order to be able to retrieve superclass constraints generically, the dictionary-passing translation of type classes stores all superclass dictionaries within the dictionary in question. In terms of the *Ord* class example, this amounts to the following definition:

$$\textbf{data}\ \mathit{OrdD}\ a = \mathit{OrdDict}\ (\mathit{EqD}\ a)\ (a \rightarrow a \rightarrow \mathit{Bool})$$

Method extraction remains the same, but of course needs to account for the additional contents of the dictionary it matches against:

$$\begin{array}{l}(\texttt{<}) :: \mathit{OrdD}\ a \rightarrow a \rightarrow a \rightarrow \mathit{Bool} \\ (\texttt{<})\ (\mathit{OrdDict}\ s\ \mathit{lt}) = \mathit{lt}\end{array}$$

Similarly, superclass extraction can be achieved by pattern matching and selection of the appropriate subterm. For example, extraction of *Eq a* from *Ord a* corresponds to the following function:

$$\begin{array}{l}\mathit{super\_ord} :: \mathit{OrdD}\ a \rightarrow \mathit{EqD}\ a \\ \mathit{super\_ord}\ (\mathit{OrdDict}\ s\ \mathit{lt}) = s\end{array}$$

Class Instances Class instances for concrete types give rise to top-level dictionary definitions. In essence, a class instance provides *proof* that a type is an instance of a class, by means of providing an implementation for the class methods. For example, the *Eq* instance for *Bool*

$$\begin{array}{l}\textbf{instance}\ \mathit{Eq}\ \mathit{Bool}\ \textbf{where} \\ \quad x\ \texttt{==}\ y\ =\ \mathit{eqBool}\ x\ y\end{array}$$

is translated to a top-level value binding *eqDBool*:

```
eqDBool :: EqD Bool
eqDBool = EqDict eqBool
```

Generic type class instances with a non-empty context give rise to term-level functions which take dictionaries as arguments and return dictionaries as a result. Let us consider the translation of the *Eq* instance for $[a]$ we presented earlier:

```
instance Eq a ⇒ Eq [a] where
  []       == []       = True
  (x : xs) == (y : ys) = (x == y) && (xs == ys)
  _        == _        = False
```

Such an instance cannot be elaborated into a dictionary: the instance declaration shows how to derive a dictionary for *Eq* $[a]$, *given that a dictionary for Eq a is available.* In short, instead of a concrete dictionary, a generic instance declaration with a non-empty context represents a *dictionary transformer*: a term-level function which transforms a set of dictionaries (instance context) into another dictionary (instance head). Thus, the above instance is elaborated into the following function:

```
eqDList :: EqD a → EqD [a]
eqDList eqda = EqDict (eqList eqda)
```

where function *eqList* is defined as follows:

```
eqList :: EqD a → [a] → [a] → Bool
eqList eqda []       []       = True
eqList eqda (x : xs) (x : xs) = (==) eqda x y && (==) (eqDList eqda) xs ys
eqList eqda _        _        = False
```

One interesting thing to notice in the above elaboration is the implementation of function *eqList*. In order to compare the elements x and y it requires a dictionary for *Eq a*, and the comparison of the sublists *xs* and *ys* requires a dictionary for *Eq* $[a]$. The latter it constructs by calling the dictionary transformer *eqDList*, effectively making the two definitions (*eqDList* and *eqList*) mutually recursive.

Finally, let us consider instances for classes with superclasses. For example, consider the *Ord* instance for *Bool*:

```
instance Ord Bool where
  (<) = ordBool
```

where *ordBool* is defined as follows (we assume *False* to be *less-than True*, since it comes first in the definition of *Bool*):

```
ordBool :: Bool → Bool → Bool
ordBool False True = True
ordBool _     _    = False
```

Since the dictionary type for *Ord* needs to store all superclass dictionaries within the newly constructed dictionary, the generated dictionary *ordDBool* takes the following form:

$$\begin{array}{l} \mathit{ordDBool} :: \mathit{OrdD}\ \mathit{Bool} \\ \mathit{ordDBool} = \mathit{OrdDict}\ \mathit{eqDBool}\ \mathit{ordBool} \end{array}$$

In this simple case the required superclass dictionary (*eqDBool*) is a top-level declaration, hence easy to retrieve. In the general case the construction of the superclass dictionaries can require arbitrary many steps. We return to this issue in Section 6.3.

Summary In summary, a class declaration with n superclass constraints and m methods gives rise to (a) a datatype declaration with a single data constructor, containing $n + m$ fields (for both superclass constraints and methods), and (b) $n + m$ extraction functions (n functions for superclasses and m for the methods). Similarly, an instance declaration with an instance context of size n gives rise to a single term-level function of arity n, taking n dictionaries as arguments and computing a dictionary as a result. This procedure is presented in formal terms in the next chapter.

Chapter 6

The Basic System

We now present a formal specification of HM with type classes, sufficiently expressive to encode a minimal version of Haskell. Simplified variants of this system have appeared in the literature, but we present its complete specification here to serve as the basis for our extensions in the forthcoming chapters (Chapters 7, 8, and 9).

As we illustrated in Section 5.4.3, the dictionary-passing elaboration of type classes requires the target language to provide support for algebraic data types and pattern matching. Even though it has been shown that (strictly-positive) data types can be encoded into System F (by means of the Böhm-Berarducci encodings (Böhm and Berarducci, 1985)), in this work we take the standard approach of extending System F with explicit support for data types and pattern matching. Hence, this section is structured as follows: Section 6.1 extends the definition of System F we provided in Section 5.2 with the additional constructs (algebraic data types, case expressions, and let-bindings). The rest of the chapter presents all formal aspects of type classes: The formalization of the basic system is spread over Sections 6.2 (syntax), 6.3 (specification of typing and elaboration), and 6.4 (type inference algorithm with elaboration). Finally, Section 6.5 states the most relevant meta-theoretical properties of the system.

6.1 Extended System F

Extending System F with data types induces changes in its syntax, typing, operational semantics, and meta-theoretical properties. We discuss each each of the changes below.

6.1.1 Syntax Extensions

Terms First, we extend the syntax of terms t to accommodate data constructors, case expressions and local let-bindings:

$$\begin{array}{lcll} t & ::= & \ldots \mid K \mid \textbf{case}\ t_1\ \textbf{of}\ \overline{p \to t_2} \mid \textbf{let}\ x : \upsilon = t_1\ \textbf{in}\ t_2 & \textit{term} \\ p & ::= & K\ \overline{x} & \textit{pattern} \end{array}$$

We allow only *simple* patterns of the form $K\ \overline{x}$, instead of *nested* patterns; case expressions with nested patterns can be desugared into nested case expressions with simple patterns (see for example the work of Maranget and Para (1994)).

Local let-bindings are not strictly necessary for our translation but since our intention is to support a sufficiently expressive source language (see Section 6.2), we avoid unnecessary desugaring by including let-bindings in our target language as well.

Types Next, we need to extend types, to accommodate type constructors:

$$\begin{array}{lcll} \upsilon & ::= & a \mid \forall a.\ \upsilon \mid T \mid \upsilon_1\ \upsilon_2 & \textit{type} \end{array}$$

Notice that function types are no longer special; they can be considered another type constructor with arity 2:

$$(\upsilon_1 \to \upsilon_2) \equiv ((\to)\ \upsilon_1)\ \upsilon_2$$

Also, by adding type application as a syntactic form we allow the partial application of type constructors (that is, to types of higher kinds). Many of the interesting examples motivating our work in the next chapters use higher-kinded types and it is desirable that they can be expressed within our system.

Declarations Finally, as we illustrated in Section 5.4.3, translating type classes requires top-level declarations: data type declarations for type classes and recursive let-bindings for class instances. We extend System F accordingly:[1]

$$\begin{array}{lcll} \textit{decl} & ::= & \textbf{let}\ x : \upsilon = t \mid \textbf{data}\ T\ \overline{a} = K_1\ \overline{\upsilon}_1 \shortmid \ldots \shortmid K_n\ \overline{\upsilon}_n & \textit{declaration} \\ \textit{pgm} & ::= & \overline{\textit{decl}} & \textit{program} \end{array}$$

[1] Notice the difference between the bars: we use $\mid$ for separating syntactic forms and $\shortmid$ for separating data constructor declarations.

6.1.2 Typing Extensions

Typing Environments Firstly, we extend typing contexts with bindings for type constructors and data constructors. Since we elide kinds altogether, we only keep track of defined type constructors but not their kind or arity (our implementations of course do):

$$\Gamma \ ::= \ \ldots \mid \Gamma, T \mid \Gamma, K : \upsilon \qquad\qquad \textit{typing environment}$$

All typing relations remain the same; we only need to add new rules for the new forms and add new typing relations for the additional constructs (patterns, declarations, and programs). We do so below.

Type Well-formedness Since we extended types with additional forms (type constructors and type applications), we need to extend the well-formedness relation to account for them. The relation still takes the form $\Gamma \vdash_{\text{TY}} \upsilon$ as in Section 5.2. The additional rules are the following:

$$\frac{T \in \Gamma}{\Gamma \vdash_{\text{TY}} T}\ \textsc{TyCon} \qquad\qquad \frac{\Gamma \vdash_{\text{TY}} \upsilon_1 \qquad \Gamma \vdash_{\text{TY}} \upsilon_2}{\Gamma \vdash_{\text{TY}} \upsilon_1\ \upsilon_2}\ \textsc{TyApp}$$

Again, since we omit kind information, both rules just check the well-formedness of the respective forms. Of course, our intention is to allow only well-kinded types; we return to this issue in Section 6.1.4.

Term Typing Typing for expressions remains the same, we only add three more rules, one for each of the new syntactic forms:

$$\frac{(K : \upsilon) \in \Gamma}{\Gamma \vdash_{\text{TM}} K : \upsilon}\ \textsc{TmCon} \qquad\qquad \frac{\Gamma \vdash_{\text{TM}} t_1 : \upsilon_1 \qquad \overline{\Gamma \vdash_{\text{P}} p \to t_2 : \upsilon_1 \to \upsilon_2}}{\Gamma \vdash_{\text{TM}} \textbf{case}\ t_1\ \textbf{of}\ \overline{p \to t_2} : \upsilon_2}\ \textsc{TmCase}$$

$$\frac{x \notin \mathit{dom}(\Gamma) \qquad \Gamma, x : \upsilon_1 \vdash_{\text{TM}} t_1 : \upsilon_1 \qquad \Gamma, x : \upsilon_1 \vdash_{\text{TM}} t_2 : \upsilon_2}{\Gamma \vdash_{\text{TM}} \textbf{let}\ x : \upsilon_1 = t_1\ \textbf{in}\ t_2 : \upsilon_2}\ \textsc{TmLet}$$

Rule TmCon looks up the type of a data constructor in the typing environment Γ. Rule TmCase handles case expressions. Once we check that the *scrutinee* has type υ_1 we check that all clauses are well-typed. This is captured in relation $\Gamma \vdash_{\text{P}} p \to t : \upsilon_1 \to \upsilon_2$, which we discuss below. Rule TmLet handles recursive let-bindings and is entirely straightforward.

Pattern Typing Pattern typing is given by relation $\Gamma \vdash_{\texttt{P}} p \to t : \upsilon_1 \to \upsilon_2$, which has a single rule:

$$\frac{\overline{x} \notin dom(\Gamma) \qquad (K : \forall \overline{a}.\ \overline{\upsilon} \to T\ \overline{a}) \in \Gamma \qquad \Gamma, \overline{x : [\overline{\upsilon}_a / \overline{a}]\upsilon} \vdash_{\texttt{TM}} t : \upsilon}{\Gamma \vdash_{\texttt{P}} K\ \overline{x} \to t : T\ \overline{\upsilon}_a \to \upsilon}\ \textsc{TmPat}$$

In short, under a typing environment Γ and a scrutinee type υ_1, we ensure that pattern p has type υ_1 and right-hand side t has type υ_2. For checking the right-hand side t, we bring the pattern variables $\overline{x}$ into scope, with the appropriate type.

Declaration Typing Finally, now our programs do not simply consist of a term, but a list of declarations. We introduce relation $\Gamma \vdash_{\texttt{D}} decl : \Gamma'$, which type-checks declarations. Given a typing environment Γ and a declaration $decl$, the resulting environment Γ' captures the typing environment extensions the declaration introduces: for a top-level binding this amounts to the term variable being bound and for a data type declaration to bindings for the type constructor and data constructors:

$$\frac{x \notin dom(\Gamma) \qquad \Gamma \vdash_{\texttt{TY}} \upsilon \qquad \Gamma, x : \upsilon \vdash_{\texttt{TM}} t : \upsilon}{\Gamma \vdash_{\texttt{D}} \mathbf{let}\ x : \upsilon = t : [x : \upsilon]}\ \textsc{TopLet}$$

$$\frac{\overline{a} \notin \Gamma \qquad \overline{\Gamma, T, \overline{a} \vdash_{\texttt{TY}} \overline{\upsilon}_i}}{\Gamma \vdash_{\texttt{D}} \mathbf{data}\ T\ \overline{a} = K_1\ \overline{\upsilon}_1 \mid \ldots \mid K_n\ \overline{\upsilon}_n : [T, K_i : \forall \overline{a}.\ \overline{\upsilon}_i \to T\ \overline{a}]}\ \textsc{TopData}$$

Rule TopLet checks recursive top-level let-bindings (x is brought in scope for checking t). Rule TopData checks data type declarations: the universally quantified constraints $\overline{a}$ are brought in scope before checking the argument types of each data constructor. The typing environment is then extended with bindings for type and data constructors.

Though not made explicit in the rule, we assume that every type constructor and data constructor is unique.

6.1.3 Operational Semantics Extensions

Finally, the syntax extensions induce a change in the operational semantics, which now needs to account for the reduction of case expressions. Relation $t_1 \longrightarrow t_2$ is extended with the two following rules:

$$\frac{t_1 \longrightarrow t_1'}{\mathbf{case}\ t_1\ \mathbf{of}\ \overline{p \to t_2} \longrightarrow \mathbf{case}\ t_1'\ \mathbf{of}\ \overline{p \to t_2}} \qquad \frac{(K_i\ \overline{x} \to t) \in \overline{p \to t_2}}{\mathbf{case}\ K_i\ \overline{t}\ \mathbf{of}\ \overline{p \to t_2} \longrightarrow [\overline{t}/\overline{x}]t}$$

The first is a congruence rule, reducing the scrutinee of a case expression. The second rule handles cases where the scrutinee is in *weak head normal form* (WHNF). In this case, the first clause that matches is selected and the pattern variables $\overline{x}$ are substituted for the actual arguments $\overline{t}$. Our notation is a bit sloppy; the delicacies of the matching order in lazy pattern matching are the subject of Part I and we do not consider them here.

6.1.4 Meta-theory Changes

We conjecture that the meta-theoretical properties of System F we presented in Section 5.2.2 are preserved under the aforementioned extensions, with the exception of strong normalization.[2] After all, this system is a subset of System F_C (Sulzmann et al., 2007a), for which the aforementioned properties are proven. Since the language now includes both (inductive) algebraic data types and general recursion (recursive let-bindings), terms are no longer strongly normalizing. Yet, this is no concern: most general-purpose languages sacrifice strong normalization for additional expressive power. As long as our target language is *type safe*, we can ensure that a well-typed elaboration has well-specified semantics.

A Note on Kinds Though we omitted all mention of kinds to avoid clutter, it is imperative (and our intention) that the system ensures the well-kindness of types. We do not want to accept non-sensical types such as $(\mathit{Int}\ \mathit{Int})$ or terms such as $(\lambda(x : \mathit{Maybe}).\ x)$. The remainder of the thesis also omits kinds (yet assuming that types are well-kinded) for the sake of brevity; we refer the reader to the work of Pierce (2002, Chapter 29) for more details on kinding, as well as the work of Jones (1993) on how to infer kinds in the presence of type classes.

6.2 Syntax

The syntax of the basic system is presented in Figure 6.1. In addition to the existing symbol classes (term variables x, y, z and type variables a, b, c), we introduce dictionary variables d and class constructors $\mathtt{TC}$. Dictionary variables are simply System F term variables but, by convention, we use x, y, z for any kind of term variable and d only for dictionary variables. A program *pgm* consists of a list of declarations *decl*, which can be class declarations *cls*, instance declarations *ins*, or value bindings *val*. The syntax of class declarations,

[2] Type safety requires all case expressions to be exhaustive, as we illustrated in Section 2.

Figure 6.1 Basic System: Syntax

$$
\begin{array}{lcll}
d & ::= & \langle \textit{dictionary variable name} \rangle & \\
\texttt{TC} & ::= & \langle \textit{class name} \rangle & \\
\\
pgm & ::= & \overline{decl} & \textit{program} \\
decl & ::= & cls \mid ins \mid val & \textit{declaration} \\
\\
cls & ::= & \textbf{class}\ \forall a.\ C \Rightarrow \texttt{TC}\ a\ \textbf{where}\ \{\ f :: \sigma\ \} & \textit{class declaration} \\
ins & ::= & \textbf{instance}\ \forall \overline{b}.\ C \Rightarrow \texttt{TC}\ \tau\ \textbf{where}\ \{\ f = e\ \} & \textit{class instance} \\
val & ::= & x = e & \textit{value binding} \\
\\
e & ::= & x \mid \lambda x.\ e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 & \textit{term} \\
\\
\tau & ::= & a \mid \tau_1 \to \tau_2 & \textit{monotype} \\
\rho & ::= & \tau \mid Q \Rightarrow \rho & \textit{qualified type} \\
\sigma & ::= & \rho \mid \forall a.\ \sigma & \textit{type scheme} \\
\\
S & ::= & \forall \overline{a}.\ C \Rightarrow \pi & \textit{constraint scheme} \\
C & ::= & \bullet \mid C, Q & \textit{constraint set} \\
Q & ::= & \pi & \textit{constraint} \\
\pi & ::= & \texttt{TC}\ \tau & \textit{class constraint} \\
\\
\Gamma & ::= & \bullet \mid \Gamma, a \mid \Gamma, x : \sigma & \textit{typing environment} \\
P & ::= & \langle \mathcal{A}_S, \mathcal{A}_I, \mathcal{C}_L \rangle & \textit{program theory}
\end{array}
$$

instances, and value bindings is standard. In order to reduce the notational burden, we omit again all mention of kinds and assume that each class has exactly one method. Additionally, we explicitly mark the type variables $\overline{a}/\overline{b}$ that are bound in the class/instance head and context.

Expressions comprise a λ-calculus, extended with recursive let bindings.

Types σ are a conservative extension of HM types (see Section 5.3.2): we interpose qualified types ρ between monotypes τ and σ. This is standard practice for HM extended with qualified types (Jones, 1992).

Next, the syntax of constraints is straightforward: constraint schemes S capture implications generated by class and instance declarations. Sets of constraints (like superclass constraints or instance contexts) are denoted by C and a single constraint is denoted by Q. For the basic system the syntax of constraints Q

coincides with class constraints π, but that is not always the case. Indeed, we lift this restriction in the upcoming chapters.

Finally, Figure 6.1 presents the syntax of typing environments Γ and program theories P. Typing environments are standard. The program theory P contains all constraint schemes generated by class and instance declarations, and gets extended with *local constraints*, when going under a qualified type.

We explicitly represent the program theory P as a triple of three constraint sets: the superclass axioms $\mathcal{A}_S$, the instance axioms $\mathcal{A}_I$ and local axioms $\mathcal{C}_L$. We use the notation $P,_{\mathrm{L}}\, d : \pi$ to denote that we extend the local component of the triple, and similar notation for the other components.

Axiom sets $\mathcal{A}$ and local axioms $\mathcal{C}$—along with other entities that we use a calligraphic font for—capture the evidence-annotated counterparts of the non-calligraphic variants:

$$\begin{array}{lcll}
\mathcal{A} & ::= & \bullet \mid \mathcal{A}, \mathcal{S} & \textit{variable-annotated constraint scheme set} \\
\mathcal{S} & ::= & d : S & \textit{variable-annotated constraint scheme} \\
\mathcal{C} & ::= & \bullet \mid \mathcal{C}, d : Q & \textit{variable-annotated constraint set} \\
\mathcal{Q} & ::= & d : Q & \textit{variable-annotated constraint} \\
\Pi & ::= & d : \pi & \textit{variable-annotated class constraint}
\end{array}$$

This allows us to present typing and elaboration succinctly in Section 6.3 below.

In earlier type class formalizations the three separate kinds of axioms that constitute the program theory P are typically conflated into a single constraint set. However, it is convenient to distinguish them for accurately stating the different restrictions imposed on them.

Moreover, the specification we present in Section 6.3 treats P as a constraint set, while the inference algorithm of Section 6.4 uses each subset differently; such a formalization is closer to actual implementations of type classes.

6.3 Typing and Elaboration into System F

Type Well-formedness and Elaboration Well-formedness and elaboration of types for the basic system extend the respective relation for HM (Figure 5.3, relation $\Gamma \vdash_{\mathrm{TY}} \sigma \rightsquigarrow v$). Since our type language now includes qualified types ρ, we simply add a new rule to handle types of this form:

$$\frac{\Gamma \vdash_{\mathrm{CT}} Q \rightsquigarrow v_1 \qquad \Gamma \vdash_{\mathrm{TY}} \rho \rightsquigarrow v_2}{\Gamma \vdash_{\mathrm{TY}} Q \Rightarrow \rho \rightsquigarrow v_1 \to v_2}\ \textsc{TyQual}$$

As we discussed in Section 5.4.3, class constraints in qualified types are interpreted as simple types in the target language (System F). Hence, the source qualification arrow "⇒" corresponds to a simple function arrow "→" in System F: a wanted class constraint is satisfied when we provide a term-level proof for it (a dictionary).

The well-formedness and elaboration of the constraint Q is captured in relation $\Gamma \vdash_{\texttt{CT}} Q \rightsquigarrow \upsilon$, given by a single rule:

$$\frac{\texttt{TC}\ \text{defined} \qquad \Gamma \vdash_{\texttt{TY}} \tau \rightsquigarrow \upsilon}{\Gamma \vdash_{\texttt{CT}} \texttt{TC}\ \tau \rightsquigarrow T_{\texttt{TC}}\ \upsilon}\ \textsc{ClsCt}$$

In essence, the *class constructor* TC is elaborated into its corresponding type constructor $T_{\texttt{TC}}$ and the class parameter is elaborated into the type constructor's type parameter. The exact definition of type $T_{\texttt{TC}}$ is generated when checking the corresponding class declaration (see Figure 6.2 below).

Term Typing and Elaboration Next, the extension of HM with type classes conservatively extends term typing. Similarly to earlier work on type class elaboration (Hall et al., 1996), the program theory P becomes part of the term typing relation (the need for this change is illustrated below). Hence, term typing for the basic system takes the form $P; \Gamma \vdash_{\texttt{TM}} e : \sigma \rightsquigarrow t$.

All rules from Figure 5.3 are parametric over the additional parameter P, so we only present the additional rules here. Similarly to Rules ($\forall I$) and ($\forall E$) that capture type abstraction introduction and elimination, Rules ($\Rightarrow I$) and ($\Rightarrow E$) capture constraint abstraction introduction and elimination, respectively:

$$\frac{\Gamma \vdash_{\texttt{CT}} Q \rightsquigarrow \upsilon \qquad d \notin \mathit{dom}(\Gamma) \cup \mathit{dom}(P) \qquad P,_{\text{L}} d : Q; \Gamma \vdash_{\texttt{TM}} e : \rho \rightsquigarrow t}{P; \Gamma \vdash_{\texttt{TM}} e : Q \Rightarrow \rho \rightsquigarrow \lambda(d : \upsilon).\ t}\ (\Rightarrow I)$$

$$\frac{P; \Gamma \vdash_{\texttt{TM}} e : Q \Rightarrow \rho \rightsquigarrow t_1 \qquad P; \Gamma \models t_2 : Q}{P; \Gamma \vdash_{\texttt{TM}} e : \rho \rightsquigarrow t_1\ t_2}\ (\Rightarrow E)$$

Rule ($\Rightarrow I$) elaborates a term with a qualified type into a λ-abstraction; the type qualification directly corresponds to a dictionary abstraction, a proof obligation that needs to be satisfied at call-sites.

Conversely, Rule ($\Rightarrow E$) eliminates an obligation, by providing a proof (dictionary) t_2 of the required class constraint. Since constraint abstraction is elaborated into a λ-abstraction, constraint elimination corresponds to term-level application. Constraint entailment with dictionary construction is captured by relation $P; \Gamma \models t : Q$, which we elaborate on below.

Finally, though not related to type classes, the basic system includes local let-bindings, handled by Rule TmLet:

$$\frac{\Gamma \vdash_{\mathtt{TY}} \tau \leadsto \upsilon \qquad \begin{array}{c} x \notin dom(\Gamma) \\ P; \Gamma, x : \tau \vdash_{\mathtt{TM}} e_1 : \tau \leadsto t_1 \end{array} \qquad P; \Gamma, x : \tau \vdash_{\mathtt{TM}} e_2 : \sigma \leadsto t_2}{P; \Gamma \vdash_{\mathtt{TM}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \sigma \leadsto \mathbf{let}\ x : \upsilon = t_1\ \mathbf{in}\ t_2}\ \textsc{TmLet}$$

In contrast to most HM-based systems in the literature (e.g., Sulzmann et al. (2007b)), let-bindings in our system are not *generalized* (that is, let-bound variables cannot have a polymorphic type). Rule TmLet internalizes this restriction by assigning x and e_1 monotype τ.

A Note on Let-generalization As Vytiniotis et al. (2010) have shown, local let-generalization disproportionally complicates type inference for its expressive power. What's more, let-generalization renders existing inference algorithms incomplete (with respect to their specification) in the presence of type equalities and local assumptions, as introduced by features like type families (Chakravarty et al., 2005a,b; Schrijvers et al., 2007, 2008), functional dependencies (Jones, 2000), and GADTs (Peyton Jones et al., 2006). Moreover, Haskell users rarely rely on let-generalization without explicit annotations (Vytiniotis et al., 2010, Section 4).

Indeed, GHC does not perform let-generalization in cases where at least one of the aforementioned features is used (Vytiniotis et al., 2011). For the same reasons, we do the same here: neither the basic system presented in this chapter or the forthcoming extensions (presented in Chapters 7, 8, and 9) consider let-generalization.[3] Nevertheless, we generalize top-level let-bindings, thus preserving the system's expressive power without complicating our presentation.

Constraint Entailment with Dictionary Construction A notion non-existent in the HM system is that of *constraint entailment*. Calling an overloaded function gives rise to *wanted* constraints, which the system should be able to automatically resolve, using *given* constraints. This procedure is captured by relation $P; \Gamma \models t : Q$. In layman's terms, under given constraints P and typing environment Γ, System F term t is a proof for constraint Q. The relation is given by a single rule:

$$\frac{(d : \forall \overline{a}.\ C \Rightarrow \mathtt{TC}\ \tau) \in P \qquad \theta = [\overline{\tau}_a / \overline{a}] \qquad P; \Gamma \models \overline{t} : \theta(C)}{P; \Gamma \models d\ \overline{\tau}_a\ \overline{t} : \mathtt{TC}\ \theta(\tau)}\ \textsc{Ent}$$

[3]Indeed, if we considered let-generalization we would face the same issues as Vytiniotis et al. (2010) in Chapter 8, where we develop a system with type classes and functional dependencies.

Figure 6.2 Basic System: Declaration Typing with Elaboration

$\boxed{\Gamma \vdash_{\texttt{CLS}} \mathit{cls} : P_S ; \Gamma_c \rightsquigarrow \overline{\mathit{decl}}}$ Class Declaration Typing

$$\frac{\begin{array}{c} \Gamma, a \vdash_{\texttt{CT}} C \rightsquigarrow \overline{\upsilon} \qquad \Gamma, a \vdash_{\texttt{TY}} \sigma \rightsquigarrow \upsilon \\ P_S = [\forall Q \in C : \ \forall a.\ \texttt{TC}\ a \Rightarrow Q] \qquad \Gamma_c = [f : \forall a.\ \texttt{TC}\ a \Rightarrow \sigma] \\ \begin{array}{lll} \overline{\mathit{decl}} & = & [\ \textbf{data}\ T_{\texttt{TC}}\ a = K_{\texttt{TC}}\ \overline{\upsilon}\ \upsilon \\ & & ,\ \textbf{let}\ d_i : \forall a.\ T_{\texttt{TC}}\ a \to \upsilon_i = \Lambda a.\ \lambda(d : T_{\texttt{TC}}\ a).\ \mathit{proj}^{i}_{\texttt{TC}}(d) \\ & & ,\ \textbf{let}\ f : \forall a.\ T_{\texttt{TC}}\ a \to \upsilon = \Lambda a.\ \lambda(d : T_{\texttt{TC}}\ a).\ \mathit{proj}^{n+1}_{\texttt{TC}}(d)] \end{array} \end{array}}{\Gamma \vdash_{\texttt{CLS}} \textbf{class}\ \forall a.\ C \Rightarrow \texttt{TC}\ a\ \textbf{where}\ \{\ f :: \sigma\ \} : P_S ; \Gamma_c \rightsquigarrow \overline{\mathit{decl}}}\ \textsc{Cls}$$

$\boxed{P ; \Gamma \vdash_{\texttt{INS}} \mathit{ins} : P_i \rightsquigarrow \mathit{decl}}$ Class Instance Typing

$$\frac{\begin{array}{c} \textbf{class}\ \forall a.\ (Q_1, \ldots, Q_n) \Rightarrow \texttt{TC}\ a\ \textbf{where}\ \{\ f :: \sigma\ \} \\ \Gamma_I = \Gamma, b \qquad P_I = P,_{\text{L}}\ \overline{d : Q}^{m} \\ \Gamma_I \vdash_{\texttt{CT}} \overline{Q}^{m} \rightsquigarrow \overline{\upsilon}^{m} \qquad \Gamma_I \vdash_{\texttt{TY}} \tau \rightsquigarrow \upsilon \qquad S = \forall \overline{b}.\ \overline{Q}^{m} \Rightarrow \texttt{TC}\ \tau \\ P_I ; \Gamma_I \models \overline{t}^{n} : \overline{[\tau/a] Q}^{n} \qquad P_I,_{\text{I}}\ d : S ; \Gamma_I \vdash_{\texttt{TM}} e : [\tau/a]\sigma \rightsquigarrow t \\ \mathit{decl} = [\textbf{let}\ d : \forall \overline{b}.\ \overline{\upsilon}^{m}_{i} \to T_{\texttt{TC}}\ \upsilon = \Lambda \overline{b}.\ \lambda(\overline{d : \upsilon}^{m}).\ K_{\texttt{TC}}\ \upsilon\ \overline{t}^{n}\ t] \end{array}}{P ; \Gamma \vdash_{\texttt{INS}} \textbf{instance}\ \forall \overline{b}.\ \overline{Q}^{m} \Rightarrow \texttt{TC}\ \tau\ \textbf{where}\ \{\ f = e\ \} : [d : S] \rightsquigarrow \mathit{decl}}\ \textsc{Ins}$$

$\boxed{P ; \Gamma \vdash_{\texttt{VAL}} \mathit{val} : \Gamma' \rightsquigarrow \mathit{decl}}$ Value Binding Typing

$$\frac{x \notin \mathit{dom}(\Gamma) \qquad P ; \Gamma, x : \sigma \vdash_{\texttt{TM}} e : \sigma \rightsquigarrow t \qquad \Gamma \vdash_{\texttt{TY}} \sigma \rightsquigarrow \upsilon}{P ; \Gamma \vdash_{\texttt{VAL}} x = e : [x : \sigma] \rightsquigarrow \textbf{let}\ x : \upsilon = t}\ \textsc{Val}$$

This method of entailment is known as *Selective Linear Definite (SLD) clause resolution* (Kowalski, 1974) or *backwards chaining*, and is the standard sound and complete resolution for Horn clauses (which, in our setting, are represented by constraint schemes S).

Essentially, Rule Ent matches the head of a given horn clause in the program theory P with the goal, and recursively entails the premises of the clause. Dictionary construction behaves accordingly: the selected dictionary transformer d is instantiated appropriately (applied to types $\overline{\tau}_a$), and then applied to the proofs for the premises $\overline{t}$.

Declaration Typing with Elaboration The specification of typing with elaboration of declarations is presented in Figure 6.2. To aid readability, we highlight all aspects of the rules that are concerned with elaboration.

Judgment $\Gamma \vdash_{\text{CLS}} cls : P_S; \Gamma_c \rightsquigarrow \overline{decl}$ handles class declarations and is given by Rule Cls. Apart from checking the well-scopedness of the class context and the method signature, it also gives rise to typing environment extension Γ_c which captures the method type, and program theory extension P_S which captures the superclass axioms we discussed in Section 5.4.2. All this information is also captured in the generated declarations $\overline{decl}$, which includes the dictionary type declaration $T_{\mathtt{TC}}$, the superclass axioms $\overline{d}^n$, and the method f. We use $proj^i_{\mathtt{TC}}(d)$ to denote the extraction of the i-th field of a class dictionary d of type $T_{\mathtt{TC}}\ a$:

$$proj^i_{\mathtt{TC}}(d) \equiv \textbf{case}\ d\ \textbf{of}\ \{\ K_{\mathtt{TC}}\ \overline{x} \rightarrow x_i\ \}$$

Judgment $P; \Gamma \vdash_{\text{INS}} ins : P_i \rightsquigarrow decl$ handles instance declarations and is also given by a single rule. Rule Ins is for the most part straightforward: we ensure that all objects are well-scoped, and additionally check (a) the entailment of superclass constraints via premise $P_I; \Gamma_I \models \overline{t}^n : \overline{[\tau/a]Q}^n$, and (b) the method implementation against its expected type via premise $P_{I,{}_I}\ d : S; \Gamma_I \vdash_{\text{TM}} e : [\tau/a]\sigma \rightsquigarrow t$. Last, the program theory extension induced by the instance (according to the logical interpretation we gave in Section 5.4.2) is captured in the scheme S, which is also elaborated into System F dictionary transformer d.

Rule Val handles top-level value bindings and is entirely straightforward.

Program Typing with Elaboration Finally, program typing combines all the above, and takes the form $\vdash_{\text{PGM}} pgm : P; \Gamma \rightsquigarrow \overline{decl}$. The judgment can be read as *"program pgm introduces typing environment* Γ*, and program theory* P*, and it gets elaborated into System F declarations* $\overline{decl}$*.* The definition of the judgment is straightforward so we omit it from our main presentation. It can be found in Appendix A.

6.4 Type Inference with Elaboration into System F

Similarly to plain HM, type inference for the basic system proceeds by first generating type constraints from the program text (constraint generation) and then solving these constraints independently of the program text (constraint solving). Yet, constraint generation for HM gives rise to wanted type equalities E, whereas the basic system needs to account for both type equalities E and sets of class constraints $\mathcal{C}$. Consequently, constraint solving for the basic system does not only result in a type substitution θ but also in an *evidence substitution*

Figure 6.3 Basic System: Term Elaboration and Constraint Generation

$\boxed{\Gamma \vdash_{\mathtt{TM}} e : \tau \rightsquigarrow t \mid \mathcal{C}; E}$ Constraint Generation

$$\frac{\overline{b}, \overline{d} \text{ fresh} \qquad (x : \forall \overline{a}.\ \overline{Q} \Rightarrow \tau) \in \Gamma \qquad \theta = [\overline{b}/\overline{a}]}{\Gamma \vdash_{\mathtt{TM}} x : \theta(\tau) \rightsquigarrow x\ \overline{b}\ \overline{d} \mid \overline{(d : \theta(Q))}; \bullet}\ \textsc{TmVar}$$

$$\frac{\begin{array}{c} a \text{ fresh} \qquad \Gamma, x : a \vdash_{\mathtt{TM}} e_1 : \tau_1 \rightsquigarrow t_1 \mid \mathcal{C}_1; E_1 \\ \Gamma, x : \tau_1 \vdash_{\mathtt{TM}} e_2 : \tau_2 \rightsquigarrow t_2 \mid \mathcal{C}_2; E_2 \qquad E = E_1, E_2, a \sim \tau_1 \end{array}}{\Gamma \vdash_{\mathtt{TM}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2 \rightsquigarrow \mathbf{let}\ x : \mathit{elab}_{\mathtt{TY}}(\tau_1) = t_1\ \mathbf{in}\ t_2 \mid (\mathcal{C}_1, \mathcal{C}_2); E}\ \textsc{TmLet}$$

$$\frac{a \text{ fresh} \qquad \Gamma, x : a \vdash_{\mathtt{TM}} e : \tau \rightsquigarrow t \mid \mathcal{C}; E}{\Gamma \vdash_{\mathtt{TM}} \lambda x.\ e : a \rightarrow \tau \rightsquigarrow \lambda(x : a).\ t \mid \mathcal{C}; E}\ \textsc{TmAbs}$$

$$\frac{a \text{ fresh} \qquad \Gamma \vdash_{\mathtt{TM}} e_1 : \tau_1 \rightsquigarrow t_1 \mid \mathcal{C}_1; E_1 \qquad \Gamma \vdash_{\mathtt{TM}} e_2 : \tau_2 \rightsquigarrow t_2 \mid \mathcal{C}_2; E_2}{\Gamma \vdash_{\mathtt{TM}} e_1\ e_2 : a \rightsquigarrow t_1\ t_2 \mid (\mathcal{C}_1, \mathcal{C}_2); (E_1, E_2, \tau_1 \sim \tau_2 \rightarrow a)}\ \textsc{TmApp}$$

η:

$\eta \ ::= \ \bullet \mid \eta \cdot [t/d]$ *evidence substitution*

Just as a type substitution θ maps type variables to monotypes, an evidence substitution η maps dictionary variables d to System F terms t (dictionaries). The remainder of Section 6.4 presents the complete type inference algorithm (constraint generation and solving) for the basic system.

Constraint Generation Constraint generation with elaboration for terms takes the form $\Gamma \vdash_{\mathtt{TM}} e : \tau \rightsquigarrow t \mid \mathcal{C} ; E$ and is presented in Figure 6.3. The judgment conservatively extends the corresponding one for HM, by generating wanted class constraints $\mathcal{C}$, alongside type equalities E. Since the differences are minimal, we highlight the parts of the rules that are specific to type classes.

Rule TmVar handles term variables. The polymorphic type $\forall \overline{a}.\ \overline{Q} \Rightarrow \tau$ of a term variable x is instantiated with fresh *unification* variables $\overline{b}$, and constraints $\overline{Q}$ are introduced as wanted constraints, instantiated likewise. In the elaborated term instantiation becomes explicit via type application. Similarly, the source-level elimination of constraints $\overline{Q}$ amounts to term-level application in System F. Arguments $\overline{d}$ capture the yet-unknown dictionaries, evidence for the wanted constraints $\overline{Q}$.

Rule TmLet handles (possibly recursive) monomorphic let-bindings. After assigning a fresh unification variable a to the term variable x, we infer types for both e_1 and e_2. Since this system does not perform let-generalization, Rule TmLet does not make a distinction between constraints generated by e_1 or e_2; they are both part of the result. Finally, we record that the (monomorphic) type of x is equal to the type of the term it is bound to: $a \sim \tau_1$.

Rule TmAbs is straightforward: the abstraction itself imposes no additional restrictions on the types, so the wanted constraints for the abstraction coincide with those for the body. Rule TmApp combines the wanted class constraints from both subterms, just as it does for wanted type equalities.

Elaboration of Types and Constraints Elaboration of types is performed by function $elab_{\text{TY}}(\sigma) = \upsilon$, given by the following clauses:

$$\begin{array}{lcl} elab_{\text{TY}}(a) & = & a \\ elab_{\text{TY}}(\tau_1 \to \tau_2) & = & elab_{\text{TY}}(\tau_1) \to elab_{\text{TY}}(\tau_2) \\ elab_{\text{TY}}(Q \Rightarrow \rho) & = & elab_{\text{CT}}(Q) \to elab_{\text{TY}}(\rho) \\ elab_{\text{TY}}(\forall a.\ \sigma) & = & \forall a.\ elab_{\text{TY}}(\sigma) \end{array}$$

The first, second, and fourth clauses are straightforward and implement the identity transformation between HM and System F types. The only case of interest is the third clause, which handles qualified types. As we informally discussed in Section 5.4.3, a qualified type in the source language corresponds to a simple arrow type in System F, from the dictionary type the class constraint represents to the elaboration of the right-hand side of the type.

The translation of a constraint to its System F counterpart is performed by function $elab_{\text{CT}}(Q) = \upsilon$, given by the following clause:

$$elab_{\text{CT}}(\mathtt{TC}\ \tau) \ = \ T_{\mathtt{TC}}\ elab_{\text{TY}}(\tau)$$

In essence, the class constructor `TC` is translated to the corresponding System F type constructor $T_{\mathtt{TC}}$ and the class parameter to a type parameter for $T_{\mathtt{TC}}$. As one might expect, each function implements the elaboration-part of the corresponding well-formedness relation (relations $\Gamma \vdash_{\text{TY}} \sigma \rightsquigarrow \upsilon$ and $\Gamma \vdash_{\text{CT}} Q \rightsquigarrow \upsilon$, respectively).

Constraint Solving The type class and equality constraints derived from terms are solved with the following two algorithms.

Solving Equality Constraints. We solve a set of equality constraints E by means of unification. Nevertheless, in contrast to plain HM, type classes

Figure 6.4 Hindley-Damas-Milner Unification Algorithm

$\boxed{unify(\overline{a}; E) = \theta_\perp}$ Type Unification Algorithm

$$
\begin{array}{lcl}
unify(\overline{a}; \bullet) & = & \bullet \\
unify(\overline{a}; E, b \sim b) & = & unify(\overline{a}; E) \\
unify(\overline{a}; E, b \sim \tau) & = & unify(\overline{a}; \theta(E)) \cdot \theta \\
\quad \text{where } b \notin \overline{a} \wedge b \notin fv(\tau) \wedge \theta = [\tau / b] & & \\
unify(\overline{a}; E, \tau \sim b) & = & unify(\overline{a}; \theta(E)) \cdot \theta \\
\quad \text{where } b \notin \overline{a} \wedge b \notin fv(\tau) \wedge \theta = [\tau / b] & & \\
unify(\overline{a}; E, (\tau_1 \to \tau_2) \sim (\tau_3 \to \tau_4)) & = & unify(\overline{a}; E, \tau_1 \sim \tau_3, \tau_2 \sim \tau_4)
\end{array}
$$

Figure 6.5 Algorithmic Constraint Entailment with Dictionary Construction

$$\boxed{\overline{a}; \mathcal{A} \models \mathcal{Q} \rightsquigarrow \mathcal{C}; \eta}$$

$$\frac{(d_I : \forall \overline{b}.\ \overline{Q}^n \Rightarrow \texttt{TC}\ \tau_2) \in \mathcal{A} \qquad unify(\overline{a}; \tau_1 \sim \tau_2) = \theta \qquad \overline{d}^n \text{ fresh}}{\overline{a}; \mathcal{A} \models d : \texttt{TC}\ \tau_1 \rightsquigarrow \overline{d_i : \theta(Q_i)}; [d_I\ \theta(\overline{b})\ \overline{d}^n / d]}\ \textsc{Ent}$$

$$\boxed{\overline{a}; \mathcal{A} \models \mathcal{C}_1 \rightsquigarrow \mathcal{C}_2; \eta}$$

$$\frac{\nexists \mathcal{Q} \in \mathcal{C} : \quad \overline{a}; \mathcal{A} \models \mathcal{Q} \rightsquigarrow \mathcal{C}'; \eta}{\overline{a}; \mathcal{A} \models \mathcal{C} \rightsquigarrow \mathcal{C}; \bullet}\ \textsc{Stop}$$

$$\frac{\overline{a}; \mathcal{A} \models \mathcal{Q} \rightsquigarrow \mathcal{C}_1; \eta_1 \qquad \overline{a}; \mathcal{A} \models \mathcal{C}, \mathcal{C}_1 \rightsquigarrow \mathcal{C}_2; \eta_2}{\overline{a}; \mathcal{A} \models \mathcal{C}, \mathcal{Q} \rightsquigarrow \mathcal{C}_2; (\eta_2 \cdot \eta_1)}\ \textsc{Step}$$

introduce explicit type annotations through method signatures. Hence, function *unify* needs to be extended to account for signatures. This is achieved by passing an additional argument: a set of "untouchable" type variables $\overline{a}$. The untouchable type variables $\overline{a}$ originate from type signatures; all other type variables are unification variables.

The updated function has signature $unify(\overline{a}; E) = \theta_\perp$ and is presented in Figure 6.4. For clarity, we highlight the differences with respect to the unification algorithm we presented in Figure 5.4. The usefulness of untouchable variables manifests itself in the third and fourth clauses, where unification is performed: unification is now allowed only if the variable to be unified is not an element of the "untouchable set". Essentially, untouchable type variables are treated by the algorithm as skolem constants and therefore cannot be substituted (they can be unified with themselves though).

Solving Type Class Constraints. Figure 6.5 defines the judgment for solving type class constraints; it takes the form $\overline{a}; \mathcal{A} \models \mathcal{C}_1 \rightsquigarrow \mathcal{C}_2; \eta$. Given a set of untouchable type variables $\overline{a}$ and an axiom set $\mathcal{A}$, it (exhaustively) replaces a set of constraints $\mathcal{C}_1$ with a set of *simpler*, residual constraints $\mathcal{C}_2$. This simplification it achieves via the auxiliary judgment $\overline{a}; \mathcal{A} \models \mathcal{Q} \rightsquigarrow \mathcal{C}; \eta$, also presented in Figure 6.5. This form differs from the specification in Section 6.3 (relation $P; \Gamma \models t : Q$) in four ways.

First, we allow constraints to be partially entailed, which in turn allows us to perform *simplification* (Jones, 1995c) of top-level signatures. This is standard practice in Haskell when inferring types. For instance, when inferring the signature for

$$f\ x = [x] == [x]$$

Haskell simplifies the derived constraint $Eq\ [a]$ to $Eq\ a$, yielding the signature $\forall a.\ Eq\ a \Rightarrow a \rightarrow Bool$.

Second, constraint simplification, as performed by the auxiliary judgment $\overline{a}; \mathcal{A} \models \mathcal{Q} \rightsquigarrow \mathcal{C}; \eta$, does not suffer from the ambiguity the specification does: instead of "guessing" a type substitution θ, we compute a unifier between the given and the wanted constraint by means of unification.

Third, evidence construction is not performed directly, by means of creating a dictionary. Instead, a dictionary substitution η is created, which maps (wanted) dictionary variables to dictionaries. The image of the mapping can refer to both dictionary variables bound by the program theory P (given), and residual (wanted) dictionary variables. This strategy is analogous to the unification algorithm, which solves type equalities by creating a type substitution for instantiating the yet-unknown types (unification variables).

Finally, in contrast to the specification, algorithmic constraint entailment does not take the complete program theory, but an axiom set. We make this design choice due to superclass constraint schemes: during simplification we do not want to replace a wanted constraint ($Eq\ a$) with a more complex ($Ord\ a$). We elaborate on the transition from the program theory P to an equally expressive axiom set $\mathcal{A}$—which does not contain superclass constraint schemes—next.

Transitive Closure of the Superclass Relation Superclass axioms often overlap with instance axioms. Consider for example the following two axioms, the first obtained by the *Eq* instance for lists and the second obtained by the *Ord* class declaration:

$$\begin{array}{ll}\forall a.\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a] & \quad\text{(a)}\\ \forall b.\ \mathit{Ord}\ b \Rightarrow \mathit{Eq}\ b & \quad\text{(b)}\end{array}$$

This is a problem for type inference, since the algorithm of Figure 6.5 would have to make a choice when faced with for example with constraint $\mathit{Eq}\ [c]$. Both axioms (a) and (b) match but to completely entail constraint $\mathit{Eq}\ [c]$ we would require $\mathit{Eq}\ c$ if we were to choose the former and $\mathit{Ord}\ [c]$ if we were to choose the latter. In order to avoid this source of non-determinism, several implementations of type classes (and notably GHC) treat superclass constraints differently.

In essence, we can pre-compute the transitive closure of the superclass relation on a set of given constraints and omit superclass axioms altogether henceforth. This procedure should also be reflected in the elaborated terms. To this end, we introduce dictionary contexts $\mathbb{E}$, that is, nested let-bindings with a hole:

$$\mathbb{E}\ ::=\ \square \mid \textbf{let}\ d : v = t\ \textbf{in}\ \mathbb{E} \qquad\qquad \textit{dictionary context}$$

Hence, during entailment we can replace the program theory P with an axiom set $\mathcal{A}$ (which does not contain any superclass axioms) and a dictionary context $\mathbb{E}$. This procedure we denote as $\textsc{ScClosure}(\overline{a}, P) = (\mathcal{A}, \mathbb{E})$:

$$\begin{array}{l}\textsc{ScClosure}(\overline{a}, \langle \mathcal{A}_S, \mathcal{A}_I, \mathcal{C}_L\rangle) = ((\mathcal{C}'_L, \mathcal{A}_I, \mathcal{C}_L), \mathbb{E})\\ \quad\text{where } (\mathcal{C}'_L, \mathbb{E}) = \mathit{mponens}^*(\overline{a}, \mathcal{A}_S, \mathcal{C}_L)\end{array}$$

Function *mponens** computes the transitive closure of the following (single-step) relation:

$$\begin{array}{l}\mathit{mponens}(\overline{a}, \mathcal{A}, d : \texttt{TC}\ \tau_1)\\ \quad = \mathit{concat}\ \{\ (d_2 : \theta(Q), \mathbb{E}) \mid (d_1 : \forall \overline{b}.\ \texttt{TC}\ \tau_1 \Rightarrow Q) \in \mathcal{A}\\ \qquad\qquad\qquad\qquad\qquad\qquad , \mathit{unify}(\overline{a}; \tau_1 \sim \tau_2) = \theta, d_2 \text{ fresh}\\ \qquad\qquad\qquad\qquad\qquad\qquad , \mathbb{E} = \textbf{let}\ d_2 : \mathit{elab}_{\text{CT}}(\theta(Q)) = d_1\ \theta(\overline{b})\ d\ \textbf{in}\ \square\ \}\end{array}$$

In layman's terms, function $\mathit{mponens}(\overline{a}, \mathcal{A}, \Pi) = (\mathcal{C}, \mathbb{E})$ tries to match the left-hand side of every available constraint scheme in $\mathcal{A}$ with the given constraint. If matching (via unification) is successful, modus ponens is used to derive the right-hand side. This procedure is also reflected in the dictionary context $\mathbb{E}$, which captures a scope where the derived dictionaries are available.

For example, if we have

$$\begin{array}{l}\overline{a}\ = m\\ \mathcal{A} = \{\ d_1 : \forall n.\ \mathit{Monad}\ n \Rightarrow \mathit{Applicative}\ n\\ \qquad ,\ d_2 : \forall k.\ \mathit{Applicative}\ k \Rightarrow \mathit{Functor}\ k\ \}\\ \Pi = d_3 : \mathit{Monad}\ m\end{array}$$

Figure 6.6 Declaration Elaboration (Selected Rules)

$\boxed{P; \Gamma \vdash_{\mathtt{INS}} \mathit{ins} : P' \rightsquigarrow \mathit{decl}}$ Class Instance Typing

$$\frac{\begin{array}{c}\textbf{class}\ \forall a.\ (Q_1, \ldots, Q_n) \Rightarrow \mathtt{TC}\ a\ \textbf{where}\ \{\ f :: \sigma\ \} \\ \Gamma_I = \Gamma, \overline{b} \qquad P_I = P,_{\mathrm{L}}\ \overline{d : Q}^m \qquad d, \overline{d}^n, \overline{d}^m\ \text{fresh} \qquad \Gamma_I \vdash_{\mathtt{CT}} \overline{Q}^m \rightsquigarrow \overline{v}^m \\ \Gamma_I \vdash_{\mathtt{TY}} \tau \rightsquigarrow v \qquad S = \forall \overline{b}.\ \overline{Q}^m \Rightarrow \mathtt{TC}\ \tau \qquad \textsc{ScClosure}(\overline{b}, P_I) = (\mathcal{A}, \mathbb{E}) \\ \overline{b}; \mathcal{A} \models \overline{d : Q}^n \rightsquigarrow \bullet; \eta \qquad \overline{b}; P_I,_{\mathrm{I}}\ d : S; \Gamma_I \vdash_{\mathtt{TM}} e : [\tau/a]\sigma \rightsquigarrow t \\ \mathit{decl} = [\textbf{let}\ d : \forall \overline{b}.\ \overline{v}_i^m \to T_{\mathtt{TC}}\ v = \Lambda \overline{b}.\ \lambda(\overline{d : v}^m).\ K_{\mathtt{TC}}\ v\ \mathbb{E}[\eta(\overline{d}^n)]\ t]\end{array}}{P; \Gamma \vdash_{\mathtt{INS}} \textbf{instance}\ \forall \overline{b}.\ \overline{Q}^m \Rightarrow \mathtt{TC}\ \tau\ \textbf{where}\ \{\ f = e\ \} : [\,d : S\,] \rightsquigarrow \mathit{decl}}\ \textsc{Ins}$$

$\boxed{P; \Gamma \vdash_{\mathtt{VAL}} \mathit{val} : \Gamma' \rightsquigarrow \mathit{decl}}$ Value Binding Typing

$$\frac{\begin{array}{c}x \notin \mathit{dom}(\Gamma) \qquad b\ \text{fresh} \qquad \Gamma, x : b \vdash_{\mathtt{TM}} e : \tau \rightsquigarrow t \mid \mathcal{C}; E \\ \mathit{unify}(\bullet; E, b \sim \tau) = \theta \qquad \overline{a} = \mathit{fv}(\theta(\mathcal{C})) \cup \mathit{fv}(\theta(\tau)) \\ \overline{a}; \mathcal{A}_I, \mathcal{C}_L \models \theta(\mathcal{C}) \rightsquigarrow \overline{d : Q}; \eta \qquad \sigma = \forall \overline{a}.\ \overline{Q} \Rightarrow \theta(\tau)\end{array}}{\langle \mathcal{A}_S, \mathcal{A}_I, \mathcal{C}_L \rangle; \Gamma \vdash_{\mathtt{VAL}} x = e : [x : \sigma] \rightsquigarrow \textbf{let}\ x : \mathit{elab}_{\mathtt{TY}}(\sigma) = \Lambda \overline{a}.\ \lambda(\overline{d : Q}).\ \mathbb{E}[\eta(\theta(t))]}\ \textsc{Val}$$

then $\mathit{mponens}^*(\overline{a}, \mathcal{A}, \Pi)$ results in the following:

$$\begin{array}{rl}\mathcal{C} & = \{\ d_4 : \mathit{Applicative}\ m, d_5 : \mathit{Functor}\ m\ \} \\ \mathbb{E} & = \textbf{let}\ d_4 : T_{\mathit{Applicative}}\ m = d_1\ m\ d_3\ \textbf{in} \\ & \quad\ \textbf{let}\ d_5 : T_{\mathit{Functor}}\ m = d_2\ m\ d_4\ \textbf{in}\ \square\end{array}$$

In plain type inference, superclass are never used; the above procedure is required in type *checking*. This is the case for method implementations, explicitly-annotated terms, and the entailment of superclass constraints in class instances.

This is better illustrated in declaration inference, which we discuss next.

Top-level Declaration Inference with Elaboration Type inference with elaboration into System F for top-level declarations is presented in Figure 6.6. Since type inference with elaboration for class declarations is identical to the specification of Figure 6.2, we only present the judgments for class instances and top-level bindings. We elaborate on both below.

Instance Inference with Elaboration Type inference for instance declarations takes the form $P; \Gamma \vdash_{\mathtt{INS}} \mathit{ins} : P' \rightsquigarrow \mathit{decl}$ and is given by Rule Ins. For the most

Figure 6.7 Subsumption Rule

$\boxed{\overline{a};\, P;\, \Gamma \vdash_{\text{TM}} e : \sigma \rightsquigarrow t}$ Explicitly-annotated Term Typing

$$\dfrac{\begin{array}{c}\Gamma \vdash_{\text{TM}} e : \tau_1 \rightsquigarrow t \mid \mathcal{C}_e;\, E_e \qquad \overline{d} \text{ fresh} \qquad \theta = \mathit{unify}(\overline{a}, \overline{b}; E_e, \tau_1 \sim \tau_2) \\ \textsc{ScClosure}(\overline{a}, (P,_{\text{L}}\, \overline{d : Q})) = (\mathcal{A}, \mathbb{E}) \qquad \overline{a}, \overline{b};\, \mathcal{A} \models \theta(\mathcal{C}_e) \rightsquigarrow \bullet;\, \eta\end{array}}{\overline{a};\, P;\, \Gamma \vdash_{\text{TM}} e : (\forall \overline{b}.\ \overline{Q} \Rightarrow \tau_2) \rightsquigarrow \Lambda \overline{b}.\ \lambda \overline{(d : \mathit{elab}_{\text{CT}}(Q))}.\ \mathbb{E}[\eta(\theta(t))]} \ (\preceq)$$

part it is identical to the corresponding rule of Figure 6.2. The most notable differences are concentrated around superclass entailment and type checking of the method implementation.

For the entailment of the superclass constraints we pre-compute the transitive closure of the superclass relation, and then (a) we generate fresh dictionary variables $\overline{d}^n$, to capture the yet-unknown superclass dictionaries, and (b) we incrementally entail the superclass constraints (requiring no residual constraints), obtaining an evidence substitution η. η maps dictionary variables $\overline{d}^n$ to generated dictionaries; the complete witness for the i-th superclass dictionary takes the form $\mathbb{E}[\eta(\theta(d_i))]$.

Method implementations have their type imposed by the method signature in the class declaration. Hence, we need to *check* rather than *infer* their type.

This operation is expressed succinctly by relation $\overline{a};\, P;\, \Gamma \vdash_{\text{TM}} e : \sigma \rightsquigarrow t$, presented in Figure 6.7. Essentially, it ensures that the inferred type for e subsumes the expected type σ. Similarly to HM (see Theorem 8), a type σ_1 is said to subsume type σ_2 if any expression that can be assigned type σ_1 can also be assigned type σ_2. In the presence of qualified types, this also requires checking that the constraints of one type can entail the constraints of the other:

$$\dfrac{P;\, \Gamma \models [\overline{\tau}/\overline{b}]\overline{Q}_2 \qquad \tau_1 = [\overline{\tau}/\overline{b}]\tau_2}{P;\, \Gamma \models (\forall \overline{a}.\ \overline{Q}_1 \Rightarrow \tau_1) \preceq (\forall \overline{b}.\ \overline{Q}_2 \Rightarrow \tau_2)}$$

Rule ($\preceq$) performs type inference and type subsumption checking simultaneously: First, it infers a monotype τ_1 for expression e, as well as wanted constraints $\mathcal{C}_e$ and type equalities E_e. Type equalities E_e should have a unifier and the inferred type τ_1 should also be unifiable with the expected type τ_2. Finally, the given constraints $\overline{Q}$ should completely entail the wanted constraints $\mathcal{C}_e$. For constraint entailment, we (again) pre-populate the given constraints with the transitive closure of the superclass axioms.

Value Binding Inference with Elaboration Finally, type inference for top-level bindings is given by Rule VAL, also in Figure 6.6. In short, the rule performs plain type-inference, simplification (Jones, 1995c) and generalization of a top-level binding.

First, we infer a monotype τ for expression e, as well as class constraints C and equality constraints E. If unification is successful, we obtain a type substitution θ, which we then apply to monotype τ and wanted class constraints C. We then simplify the wanted constraints via constraint entailment to a set of simpler constraints $\overline{Q}$; since there are no local constraints, superclass axioms are ignored. For simplicity, we do not utilize *interaction rules* (e.g., we do not simplify the constraints $\{Eq\ a, Ord\ a\}$ to $\{Ord\ a\}$), but it is straightforward to do so. Finally, type variables $\overline{a}$ that have not been unified are generalized. All the above reasoning is directly reflected in the elaboration of the declaration, which we highlight to aid readability.

Type-Annotated Value Binding Inference with Elaboration Though the syntax of Figure 6.1 does not allow explicitly-annotated top-level bindings, it is instructive to consider such an extension. Such a declaration is straightforward to check:

$$\frac{\bullet; P; \Gamma, x : \sigma \vdash_{\text{TM}} e : \sigma \rightsquigarrow t}{P; \Gamma \vdash_{\text{VAL}} \textbf{let}\ x : \sigma = e : [x : \sigma] \rightsquigarrow \textbf{let}\ x : elab_{\text{TY}}(\sigma) = t}\ \textsc{ValSig}$$

In case the expression is explicitly annotated, type inference directly corresponds to an inference-and-subsumption-check, as given by judgment in Figure 6.7 above.

Program Typing with Elaboration Program typing with elaboration is identical to its specification and is thus omitted.

6.5 Meta-theoretical Properties

In order to ensure well-specified semantics for type classes, it is common to impose several external restrictions on their usage. Under these conditions, we conjecture that all properties that hold for the HM system (see Section 5.3.3) can be proven. Hence, we discuss here only the two properties that are more challenging in the presence of type classes: termination and coherence of elaboration.

Termination of Type Inference First, for decidable type inference it is required that type inference terminates on all inputs. This property is ensured under the *Termination Conditions*:

(a) The superclass relation forms a *directed acyclic graph* (DAG).

(b) In each class instance (**instance** $\forall \bar{b}.\ C \Rightarrow \texttt{TC}\ \tau$):

- no variable has more occurrences in a type class constraint in the instance context C than the head ($\texttt{TC}\ \tau$), and
- each class constraint in the instance context C has fewer constructors and variables (taken together, counting repetitions) than the head ($\texttt{TC}\ \tau$).

The first restriction ensures that the computation of the transitive closure of the superclass relation (as performed by function $\textsc{ScClosure}(\bar{a}, P)$) is terminating.

The second restriction, borrowed from the Paterson Conditions (Sulzmann et al., 2007b, Def. 11), ensures that instance contexts are *decreasing*, so that class resolution is also terminating.

By introducing some additional notation, we can formalize Condition (b) on constraint schemes derived by instance declarations as follows:

$$\frac{\|Q_i\| < \|Q\| \qquad \forall a \in \bar{a} :\ occ_a(Q_i) \leq occ_a(Q)}{term(\forall \bar{a}.\ (Q_1, \ldots, Q_n) \Rightarrow Q)}$$

where function $\|\cdot\|$ computes the size of a constraint and $occ_a(\cdot)$ computes the number of occurrences of variable a in a constraint. The size ($\|\tau\| = \mathbb{N}$) and the number of occurrences of a in a monotype ($occ_a(\tau) = \mathbb{N}$) are given by the following functions (and are straightforwardly generalized to constraints):

$$\begin{array}{lcl} \|a\| & = & 1 \\ \|\tau_1 \rightarrow \tau_2\| & = & 1 + \|\tau_1\| + \|\tau_2\| \end{array} \qquad \begin{array}{lcl} occ_a(b) & = & \left\{ \begin{array}{ll} 1 & \text{, if } a = b \\ 0 & \text{, if } a \neq b \end{array} \right. \\ occ_a(\tau_1 \rightarrow \tau_2) & = & occ_a(\tau_1) + occ_a(\tau_2) \end{array}$$

Coherence Conditions Coherence—which means that every different valid type derivation for a program leads to a resulting program that has the same dynamic semantics—is ensured if we restrict ourselves to *non-overlapping instances* and *non-ambiguous types*:

Definition 3 (Non-overlapping Instances)**.** *Any two instance heads* ($\mathit{TC}\ \tau_1$) *and* ($\mathit{TC}\ \tau_2$) *for the same class should not overlap* ($\nexists\theta.\ \theta(\tau_1) = \theta(\tau_2)$)*.*

The key idea behind this requirement is that computational content can only originate from class instances; if no two instances overlap, multiple derivations may exist but their semantics have to be the same.

Definition 4 (Non-Ambiguous Types and Schemes)**.** *A well-formed type $\sigma = \forall \overline{a}.\ C \Rightarrow \tau$ is unambiguous (denoted by $\mathit{unamb}(\sigma)$) iff $\mathit{fv}(C) \subseteq \mathit{fv}(\tau)$. Similarly, a well-formed constraint scheme $S = \forall \overline{a}.\ C \Rightarrow \pi$ is unambiguous (denoted by $\mathit{unamb}(S)$) iff $\mathit{fv}(C) \subseteq \mathit{fv}(\pi)$.*

Essentially, Haskell '98 requires that all quantified type variables in a qualified type or a constraint scheme appear in the head. Thus, the non-ambiguity requirement ensures that the inference algorithm does not guess any polymorphic instantiations, since different choices lead to different proofs. Consider the following instances:

$$\begin{array}{l}
\mathbf{instance}\ C\ \mathit{Char}\ \mathbf{where}\ \{\ \ldots\ \} \\
\mathbf{instance}\ C\ \mathit{Bool}\ \ \mathbf{where}\ \{\ \ldots\ \} \\
\mathbf{instance}\ C\ a \Rightarrow D\ \mathit{Int}\ \mathbf{where}\ \{\ \ldots\ \}
\end{array}$$

The third instance gives rise to the axiom $\forall a.\ C\ a \Rightarrow D\ \mathit{Int}$. When resolving $D\ \mathit{Int}$ with this axiom we can choose a to be either *Char* or *Bool* and thus select a different C instance.

A similar issue arises with ambiguous type signatures. Consider the folklore *read*/*show* example:

$$\begin{array}{l}
h :: (\mathit{Read}\ a, \mathit{Show}\ a) \Rightarrow \mathit{String} \to \mathit{String} \\
h\ x = \mathit{show}\ (\mathit{read}\ x)
\end{array}$$

Whether the signature for h is provided by the programmer or inferred by the inference algorithm is irrelevant; unless a concrete type for $(\mathit{read}\ x)$ is explicitly specified, h is ambiguous (a can be chosen to be *Int*, *Bool*, and so on).

6.6 Scientific Output

In this chapter we have presented a formalization of type classes, including a specification of typing and elaboration, as well as a type inference with elaboration algorithm. The main contribution of this chapter is the formalization of superclasses, an important—yet often neglected—aspect of type classes. We believe that this system is a great candidate for proving meta-theoretical properties of type classes, in a way that also reflects their implementations. The remainder of Part II develops three type class extensions, building upon the specification of this system and referring to this chapter for all relevant notions.

Chapter 7

Quantified Constraints

In this chapter we present the first of our extensions to type classes: *Quantified Class Constraints.*

The chapter is structured as follows: Section 7.1 motivates the development of this heavily requested feature. Through a series of examples, we present the expected semantics of quantified constraints, as well the benefits a programmer can expect from their use. Section 7.2 presents the extensions the feature induces to the declarative specification of the basic system we presented in Section 6.3. Section 7.3 presents the extensions to the type inference and elaboration algorithm, again with respect to the corresponding algorithm for the basic system (Section 6.4). Section 7.4 discusses the meta-theoretical properties of the extended system, and especially the conditions we require in order to ensure the termination of the type inference algorithm. Section 7.5 discusses related work and encodings of quantified constraints in the literature. Finally, Section 7.6 concludes and summarizes the scientific output of this work.

7.1 Motivation

7.1.1 A History of Quantified Constraints

Ever since type classes were introduced in Haskell (Wadler and Blott, 1989), they have been the subject of numerous extensions that increase their expressive power and enable new applications. Several of these implemented extensions were inspired by the analogy between type classes and Horn clauses (see Section 5.4.2).

Yet, Horn clauses have their limitations. As a small side-product of their work on derivable type classes, Hinze and Peyton Jones (2000) have proposed to raise the expressive power of type classes to essentially the universal fragment of Hereditary Harrop logic (Harrop, 1956) with what they call *quantified class constraints*. Their motivation was to deal with higher-kinded types which seemed to require instance declarations that were impossible to express in the type-class system of Haskell at that time.

Unfortunately, Hinze and Peyton Jones never did elaborate on quantified class constraints. Later, Lämmel and Peyton Jones (2003) found a workaround for the particular problem of the derivable type classes work that did not involve quantified class constraints. Nevertheless the idea of quantified class constraints has whet the appetite of many researchers and developers. GHC ticket #2893[1], requesting for quantified class constraints, was opened in 2008 and is still open today. Commenting on this ticket in 2009, Peyton Jones states that *"their lack is clearly a wart, and one that may become more pressing"*, yet clarifies in 2014 that *"[t]he trouble is that I don't know how to do type inference in the presence of polymorphic constraints."* In 2010, 10 years after the original idea, Hinze (2010) rues that the feature has not been implemented yet. As recently as 2016, Chauhan et al. (2016) regret that *"Haskell does not allow the use of universally quantified constraints"* and in 2017 Spivey (2017) has to use pseudo-Haskell when modeling with quantified class constraints. While various workarounds have been proposed and are used in practice (Trifonov, 2003; Schrijvers and Oliveira, 2011; Kmett, 2017), none has stopped the clamor for proper quantified class constraints. This chapter finally elaborates the original idea of quantified class constraints into a fully fledged language design.

The remainder of this section introduces quantified class constraints and illustrates the expressive power afforded by quantified class constraints to capture several requirements of type class instances more succinctly, and to provide terminating resolution for a large group of applications.

7.1.2 Precise and Succinct Specifications

Monad Transformers Consider the type class for monad transformers (Jones, 1995a) as defined in the *Monad Transformer Library* (MTL):[2]

$$\begin{array}{l}\textbf{class}\ \mathit{Trans}\ t\ \textbf{where}\\ \quad \mathit{lift} :: \mathit{Monad}\ m \Rightarrow m\ a \rightarrow (t\ m)\ a\end{array}$$

[1] `https://ghc.haskell.org/trac/ghc/ticket/2893`
[2] `https://hackage.haskell.org/package/mtl`

What is not formally expressed in the above type class declaration, but implicitly expected, is that for any type T that instantiates *Trans* there should also be a *Monad* instance of the form:

$$\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (T\ m)\ \textbf{where} \ldots$$

Because the type checker is not told about this requirement, it will not accept the following definition of monad transformer composition.

$$\textbf{newtype}\ (t_1 * t_2)\ m\ a = C\ \{\ \mathit{runC} :: t_1\ (t_2\ m)\ a\ \}$$

$$\begin{array}{l}\textbf{instance}\ (\mathit{Trans}\ t_1, \mathit{Trans}\ t_2) \Rightarrow \mathit{Trans}\ (t_1 * t_2)\ \textbf{where} \\ \quad \mathit{lift} = C \cdot \mathit{lift} \cdot \mathit{lift}\end{array}$$

The idea of this code is to *lift* from monad m to $(t_2\ m)$ and then to *lift* from $(t_2\ m)$ to $t_1\ (t_2\ m)$. However, the second *lift* is only valid if $(t_2\ m)$ is a monad and the type checker has no way of establishing that this fact holds for all monad transformers t_2. Workarounds for this problem do exist in current Haskell (Trifonov, 2003; Jaskelioff, 2011; Schrijvers and Oliveira, 2011), but they clutter the code with heavy encodings.

Quantified class constraints allow us to state this requirement explicitly as part of the *Trans* class declaration:

$$\begin{array}{l}\textbf{class}\ (\forall m.\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (t\ m)) \Rightarrow \mathit{Trans}\ t\ \textbf{where} \\ \quad \mathit{lift} :: \mathit{Monad}\ m \Rightarrow m\ a \to (t\ m)\ a\end{array}$$

The instance for transformer composition $t_1 * t_2$ now typechecks.

Second-Order Functors Another example can be found in the work of Hinze (2010). He represents parameterized datatypes, like polymorphic lists and trees, as the fixpoint *Mu* of a *second-order functor*:

$$\textbf{data}\ \mathit{Mu}\ h\ a = \mathit{In}\ \{\ \mathit{out} :: h\ (\mathit{Mu}\ h)\ a\ \}$$

$$\textbf{data}\ \mathit{List}_2\ f\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (f\ a)$$

$$\textbf{type}\ \mathit{List} = \mathit{Mu}\ \mathit{List}_2$$

A second-order functor h is a type constructor that sends functors to functors. This can be concisely expressed with the quantified class constraint $\forall f.\ \mathit{Functor}\ f \Rightarrow \mathit{Functor}\ (h\ f)$, for example in the *Functor* instance of *Mu*:

$$\begin{array}{l}\textbf{instance}\ (\forall f.\ \mathit{Functor}\ f \Rightarrow \mathit{Functor}\ (h\ f)) \Rightarrow \mathit{Functor}\ (\mathit{Mu}\ h)\ \textbf{where} \\ \quad \mathit{fmap}\ f\ (\mathit{In}\ x) = \mathit{In}\ (\mathit{fmap}\ f\ x)\end{array}$$

Although this is Hinze's preferred formulation he remarks that:

> *Unfortunately, the extension has not been implemented yet. It can be simulated within Haskell '98 (Trifonov, 2003), but the resulting code is somewhat clumsy.*

Johann and Ghani (2008) use essentially the same data-generic representation, the fixpoint of second-order functors, to represent so-called *nested datatypes* (Bird and Meertens, 1998). For instance, Hinze (2000) represents perfect binary trees with the nested datatype

$$\textbf{data}\ \mathit{Perfect}\ a = \mathit{Zero}\ a \mid \mathit{Succ}\ (\mathit{Perfect}\ (a, a))$$

This can be expressed with the generic representation as *Mu HPerf*, the fixpoint of the second-order functor *HPerf*, defined as

$$\textbf{data}\ \mathit{HPerf}\ f\ a = \mathit{HZero}\ a \mid \mathit{HSucc}\ (f\ (a, a))$$

Johann and Ghani's notion of second-order functor differs slightly from Hinze's.[3] Ideally, their notion would be captured by the following class declaration:

$$\begin{array}{l}\textbf{class}\ (\forall f.\ \mathit{Functor}\ f \Rightarrow \mathit{Functor}\ (h\ f)) \Rightarrow \mathit{HFunctor}\ h\ \textbf{where}\\ \quad \mathit{hfmap} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow (\forall x.\ f\ x \to g\ x) \to (\forall x.\ h\ f\ x \to h\ g\ x)\end{array}$$

Like in Hinze's case, the quantified class constraint expresses that a second-order functor takes first-order functors to first-order functors. Additionally, second-order functors provide a second-order *fmap*, called *hfmap*, which replaces f by g, to take values of type $(h\ f\ x)$ to type $(h\ g\ x)$. Yet, in the absence of actual support for quantified class constraints, Johann and Ghani provide the following declaration instead:

$$\begin{array}{l}\textbf{class}\ \mathit{HFunctor}\ h\ \textbf{where}\\ \quad \mathit{ffmap}\ :: \mathit{Functor}\ f \Rightarrow (a \to b) \to (h\ f\ a \to h\ f\ b)\\ \quad \mathit{hfmap} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow (\forall x.\ f\ x \to g\ x) \to (\forall x.\ h\ f\ x \to h\ g\ x)\end{array}$$

In essence, they inline the *fmap* method provided by the quantified class constraint in the *HFunctor* class. This is unfortunate because it duplicates the *Functor* class's functionality.

7.1.3 Terminating Corecursive Resolution

Quantified class constraints were first proposed by Hinze and Peyton Jones (2000) as a solution to a problem of diverging type class resolution. Consider their generalized rose tree datatype

$$\textbf{data}\ \mathit{GRose}\ f\ a = \mathit{GBranch}\ a\ (f\ (\mathit{GRose}\ f\ a))$$

[3] It is more in line with the category theoretical notion of endofunctors over the category of endofunctors.

and its *Show* instance

$$\begin{array}{l}\textbf{instance}\ (\mathit{Show}\,a, \mathit{Show}\,(f\ (\mathit{GRose}\,f\,a))) \Rightarrow \mathit{Show}\,(\mathit{GRose}\,f\,a)\ \textbf{where}\\ \quad \mathit{show}\ (\mathit{GBranch}\ x\ \mathit{xs}) = \mathit{unwords}\ [\mathit{show}\ x, \texttt{"}-\texttt{"}, \mathit{show}\ \mathit{xs}]\end{array}$$

Notice the two constraints in the instance context which are due to the two *show* invocations in the method definition. Standard recursive type class resolution would diverge when faced with the constraint (*Show* (*GRose* [] *Bool*)). Indeed, it would recursively resolve the instance context: *Show Bool* is easily dismissed, but *Show* [*GRose* [] *a*] requires resolving *Show* (*GRose* [] *Bool*) again. Clearly this process loops.

To solve this problem, Hinze and Peyton Jones proposed to write the *GRose* instance with a quantified type class constraint as:

$$\begin{array}{l}\textbf{instance}\ (\mathit{Show}\,a, \forall x.\ \mathit{Show}\,x \Rightarrow \mathit{Show}\,(f\,x)) \Rightarrow \mathit{Show}\,(\mathit{GRose}\,f\,a)\ \textbf{where}\\ \quad \mathit{show}\ (\mathit{GBranch}\ x\ \mathit{xs}) = \mathit{unwords}\ [\mathit{show}\ x, \texttt{"}-\texttt{"}, \mathit{show}\ \mathit{xs}]\end{array}$$

This would avoid the diverging loop in the type system extension they sketch, because the two recursive resolvents, *Show Bool* and $\forall x.$ *Show* $x \Rightarrow$ *Show* $[x]$ are readily discharged with the available *Bool* and $[a]$ instances.

When faced with the same looping issue in their *Scrap Your Boilerplate* work, Lämmel and Peyton Jones (2005) implemented a different solution: *cycle-aware constraint resolution.* This approach detects that a recursive resolvent is identical to one of its ancestors and then ties the (co-)recursive knot at the level of the underlying type class dictionaries.

Unfortunately, cycle-aware resolution is not a panacea. It only deals with a particular class of diverging resolutions, those that cycle. The fixpoint of the second-order functor *HPerf* presented above is beyond its capabilities.

$$\begin{array}{l}\textbf{instance}\ (\mathit{Show}\ (h\ (\mathit{Mu}\ h)\ a)) \Rightarrow \mathit{Show}\ (\mathit{Mu}\ h\ a)\ \textbf{where}\\ \quad \mathit{show}\ (\mathit{In}\ x) = \mathit{show}\ x\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (\mathit{Show}\ a, \mathit{Show}\ (f\ (a,a))) \Rightarrow \mathit{Show}\ (\mathit{HPerf}\ f\ a)\ \textbf{where}\\ \quad \mathit{show}\ (\mathit{HZero}\ a)\ \ = \texttt{"}(Z\ \ \texttt{"} +\!\!+ \mathit{show}\ a +\!\!+ \texttt{")"}\\ \quad \mathit{show}\ (\mathit{HSucc}\ \mathit{xs}) = \texttt{"}(S\ \ \texttt{"} +\!\!+ \mathit{show}\ \mathit{xs} +\!\!+ \texttt{")"}\end{array}$$

Resolving *Show* (*Mu HPerf Int*) diverges without cycling back to the original constraint due to the nestedness of the perfect tree type:

$$\begin{array}{ll} & \underline{\mathit{Show}\ (\mathit{Mu}\ \mathit{HPerf}\ \mathit{Int})}\\ \rightarrowtail & \mathit{Show}\ (\mathit{HPerf}\ (\mathit{Mu}\ \mathit{HPerf})\ \mathit{Int})\\ \rightarrowtail & \mathit{Show}\ \mathit{Int}, \underline{\mathit{Show}\ (\mathit{Mu}\ \mathit{HPerf}\ (\mathit{Int},\mathit{Int}))}\\ \rightarrowtail & \mathit{Show}\ (\mathit{HPerf}\ (\mathit{Mu}\ \mathit{HPerf})\ (\mathit{Int},\mathit{Int}))\\ \rightarrowtail & \mathit{Show}\ (\mathit{Int},\mathit{Int}), \underline{\mathit{Show}\ (\mathit{Mu}\ \mathit{HPerf}\ ((\mathit{Int},\mathit{Int}),(\mathit{Int},\mathit{Int})))}\\ \rightarrowtail & \ldots\end{array}$$

Figure 7.1 Quantified Class Constraints: Syntax Extensions

$$\begin{array}{rcll} Q & ::= & \pi \mid Q_1 \Rightarrow Q_2 \mid \forall a.\ Q & \textit{constraint} \\ P & ::= & \langle \mathcal{C}_S, \mathcal{C}_I, \mathcal{C}_L \rangle & \textit{program theory} \end{array}$$

In contrast, with quantified type class constraints we can formulate the instances in a way that resolution does terminate.

instance $(\mathit{Show}\, a, \forall f.\ \forall x.\ (\mathit{Show}\, x, \forall y.\ \mathit{Show}\, y \Rightarrow \mathit{Show}\, (f\, y)) \Rightarrow \mathit{Show}\, (h\, f\, x))$
  $\Rightarrow \mathit{Show}\, (\mathit{Mu}\, h\, a)$ **where**
  $\mathit{show}\, (\mathit{In}\, x) = \mathit{show}\, x$

instance $(\mathit{Show}\, a, \forall x.\ \mathit{Show}\, x \Rightarrow \mathit{Show}\, (f\, x)) \Rightarrow \mathit{Show}\, (\mathit{HPerf}\, f\, a)$ **where**
  $\mathit{show}\ (\mathit{HZero}\ a)\ = \mathit{unwords}\ [\texttt{"(}Z\texttt{"}, \mathit{show}\ a, \texttt{")"}]$
  $\mathit{show}\ (\mathit{HSucc}\ \mathit{xs}) = \mathit{unwords}\ [\texttt{"(}S\texttt{"}, \mathit{show}\ \mathit{xs}, \texttt{")"}]$

7.1.4 Summary

In summary, quantified type class constraints enable (a) expressing more of a type class's specification in a natural and succinct manner, and (b) terminating type class resolution for a larger group of applications.

In the remainder of this chapter we discuss all formal aspects of the development of quantified class constraints.

7.2 Declarative Specification

This section presents the changes to the specification (syntax, typing, and elaboration) of type classes (Chapter 6) by the introduction of quantified constraints.

7.2.1 Syntax

The syntax of type classes with quantified constraints is—for the most part—identical to the one for the basic system (Section 6.2, Figure 6.1). The differences between the two are highlighted in Figure 7.1.

Our calculus differs from Haskell '98 in that it conservatively generalizes the language of constraints. In Haskell '98 the constraints that can appear in type signatures and in class and instance contexts are simple class constraints π of the form $\texttt{TC}\ \tau$. As a consequence, the constraint schemes or axioms that are derived from instances (and for superclasses) are Horn clauses of the form:[4]

$$\forall \overline{a}.\ (\pi_1 \wedge \ldots \wedge \pi_n) \Rightarrow \pi_0$$

These axioms are similar to rank-1 polymorphic types in the sense that the quantifiers (and the implication) only occur on the outer level. We allow a more general form of constraints Q where, in analogy with higher-rank types, quantifiers and implications occur in nested positions. This more expressive form of constraints can occur in signatures and class/instance contexts. Consequently, the syntactic sort Q of constraints and axioms S is one and the same.

This allows us to represent the program theory P as three lists of variable-annotated constraints, instead of the more restrictive form of three lists of variable-annotated constraint schemes of Figure 6.1. In Haskell, the local constraints are basic type class constraints π only, while the instance and superclass axioms have the more expressive Horn clause form (constraint schemes S). In contrast, in our setting all three components support the same (and more general) form of *Harrop formulae.* Thus, constraint schemes S do not appear in the remainder of this chapter; they are completely subsumed by constraints Q.

Lastly, note that constraint schemes of the form $\forall \overline{a}.\ (Q_1 \wedge \ldots \wedge Q_n) \Rightarrow Q_0$, used in earlier formalizations of type classes (e.g., by Morris (2014)), are not valid syntax for our constraints Q because we do not provide a notation for conjunction. Yet, we can easily see the scheme notation as syntactic sugar for a curried representation:

$$\forall \overline{a}.\ (Q_1 \wedge \cdots \wedge Q_n) \Rightarrow Q_0 \quad \equiv \quad \forall \overline{a}.\ Q_1 \Rightarrow (\ldots (Q_n \Rightarrow Q_0) \ldots)$$

7.2.2 The Type System

Since the differences between the basic system and the one extended with quantified constraints are concentrated around the shape of constraints, most typing relations (as well as the elaboration aspects of them) remain the same: term typing, well-formedness of types, declaration typing, and program typing are identical to the ones we presented in Section 6.3. Hence, we refer the reader to Section 6.3 for their definition.

[4]Which we capture in the syntax of the basic system by constraint schemes S.

The relations that are affected by the genericity of the new constraints Q are the well-formedness of constraints and, more notably, constraint entailment. We discuss each in detail below.

Constraint Well-formedness with Elaboration Well-formedness and elaboration of constraints ($\Gamma \vdash_{\text{CT}} Q \rightsquigarrow \upsilon$) is extended with two more rules, to account for the new syntactic forms. It is given by the following rules:

$$\frac{\texttt{TC}\text{ defined} \qquad \Gamma \vdash_{\text{TY}} \tau \rightsquigarrow \upsilon}{\Gamma \vdash_{\text{CT}} \texttt{TC}\ \tau \rightsquigarrow T_{\texttt{TC}}\ \upsilon}\ (\text{C}\pi) \qquad \frac{\Gamma \vdash_{\text{CT}} Q_1 \rightsquigarrow \upsilon_1 \qquad \Gamma \vdash_{\text{CT}} Q_2 \rightsquigarrow \upsilon_2}{\Gamma \vdash_{\text{CT}} Q_1 \Rightarrow Q_2 \rightsquigarrow \upsilon_1 \rightarrow \upsilon_2}\ (\text{C}{\Rightarrow})$$

$$\frac{a \notin \Gamma \qquad \Gamma, a \vdash_{\text{CT}} Q \rightsquigarrow \upsilon}{\Gamma \vdash_{\text{CT}} \forall a.\ Q \rightsquigarrow \forall a.\ \upsilon}\ (\text{C}\forall)$$

Rule (Cπ) handles simple class constraints and is identical to the one we presented in Section 6.3. Rule (C$\Rightarrow$) handles implication constraints and Rule (C$\forall$) constraints with universal quantification. An interesting takeaway from the above rules is the elaboration technique they hint at: universal quantification in constraints corresponds to universally quantified types in System F and constraint implication corresponds to λ-abstraction. Hence, in the extended setting of quantified class constraints, constraints are not elaborated to dictionaries but *dictionary transformers*.

Additionally, notice that arbitrary nested universal quantification in constraints translates to higher-rank types in System F. Though this might seem problematic for type inference (as we discussed in Section 5.3.1), quantified constraints are always explicitly specified in class and instance declarations. Thus, type inference remains decidable.

7.2.3 Constraint Entailment

Following the approach of Schrijvers et al. (2017) for their COCHIS calculus, we present constraint entailment in two steps. First, we provide an easy-to-understand and expressive, yet also highly ambiguous, specification. Then we present a syntax-directed, semi-algorithmic variant that takes the ambiguity away, but has a more complicated formulation inspired by the *focusing* technique used in proof search (Miller et al., 1989; Andreoli, 1992; Liang and Miller, 2009).

Declarative Specification Constraint entailment takes the standard form $P; \Gamma \models t : Q$, and its high-level declarative specification is given by the rules

Figure 7.2 Constraint Entailment: Declarative Specification

$\boxed{P; \Gamma \models t : Q}$ Constraint Entailment Specification

$$\frac{(d : Q) \in P}{P; \Gamma \models d : Q}\ \textsc{SpecC} \qquad \frac{a \notin \Gamma \qquad P; \Gamma, a \models t : Q}{P; \Gamma \models \Lambda a.\ t : \forall a.\ Q}\ (\forall\text{IC})$$

$$\frac{d \notin dom(P) \cup dom(\Gamma) \qquad \Gamma \vdash_{\text{CT}} Q_1 \rightsquigarrow \upsilon \qquad P, d : Q_1; \Gamma \models t : Q_2}{P; \Gamma \models \lambda(d : \upsilon).\ t : Q_1 \Rightarrow Q_2}\ (\Rightarrow\text{IC})$$

$$\frac{P; \Gamma \models t : \forall a.\ Q \qquad \Gamma \vdash_{\text{TY}} \tau \rightsquigarrow \upsilon}{P; \Gamma \models t\ \upsilon : [\tau/a]\, Q}\ (\forall\text{EC})$$

$$\frac{P; \Gamma \models t_1 : Q_1 \Rightarrow Q_2 \qquad P; \Gamma \models t_2 : Q_1}{P; \Gamma \models t_1\ t_2 : Q_2}\ (\Rightarrow\text{EC})$$

we present in Figure 7.2. To aid readability, we separate the entailment aspect of the specification from dictionary construction by highlighting the latter.

If we interpret constraints Q as logical formulas, the rules of Figure 7.2 are nothing more than the rules of the universal fragment of Hereditary Harrop logic (Harrop, 1956). Rule SpecC is the standard axiom rule. Rules ($\Rightarrow$IC) and ($\Rightarrow$EC) correspond to implication introduction and elimination, respectively. Similarly, Rules ($\forall$IC) and ($\forall$EC) correspond to introduction and elimination of universal quantification, respectively. These are also essentially the rules Hinze and Peyton Jones (2000) propose.

The dictionary-constructing aspect of the rules follows directly if one interprets the universal fragment of Hereditary Harrop logic constructively: dictionary construction is syntax-directed on the dictionary t, corresponding one-to-one to term typing for the polymorphic λ-calculus (Section 5.2, Figure 5.1).

Unfortunately, while compact and elegant, there is a serious downside to these rules: they are highly ambiguous and give rise to many trivially different proofs for the same constraint. For instance, assuming $\Gamma = \bullet, a$ and $P = \langle \bullet, \bullet, Eq\ a \rangle$, here are only two of the infinitely many proofs of $P; \Gamma \models Eq\ a$:[5]

$$\frac{Eq\ a \in P}{P; \Gamma \models Eq\ a}\ \textsc{SpecC}$$

[5]We omit for brevity the corresponding dictionaries; they do not contribute anything to understanding the issue at hand.

versus

$$\dfrac{\dfrac{\dfrac{Eq\ a \in P'}{P'; \Gamma \models Eq\ a}\ \textsc{SpecC}}{P; \Gamma \models Eq\ a \Rightarrow Eq\ a}\ (\Rightarrow\text{IC}) \qquad \dfrac{Eq\ a \in P}{P; \Gamma \models Eq\ a}\ \textsc{SpecC}}{P; \Gamma \models Eq\ a}\ (\Rightarrow\text{EC})$$

where $P' = P,_{\text{L}}\ Eq\ a$. Observe that the latter proof makes an unnecessary appeal to implication introduction.

Type-directed Specification To avoid the trivial forms of ambiguity like in the example, we adopt a solution from proof search known as *focusing* (Andreoli, 1992). This solution was already adopted by the Cochis calculus, for the same reason. The key idea of focusing is to provide a syntax-directed definition of constraint entailment where only one inference rule applies at any given time.

Figure 7.3 presents our definition of constraint entailment with focusing. The main judgment $P; \Gamma \models t : Q$ is defined in terms of two auxiliary judgments, $P; \Gamma \models t : [Q]$ and $\Gamma; t : [Q] \models t' : \pi \rightsquigarrow \mathcal{C}$. In the remainder of this paragraph we focus on the entailment aspect of the rules only; the dictionary construction aspect is discussed in the next paragraph.

The main entailment judgment is equivalent to the first auxiliary judgment $P; \Gamma \models t : [Q]$. This auxiliary judgment focuses on the constraint Q whose entailment is checked—we call this constraint the “goal”. There are three rules, for the three possible syntactic forms of Q. Rules ($\Rightarrow$R) and ($\forall$R) decompose the goal by applying implication and quantifier introductions respectively. Once the goal is stripped down to a simple class constraint π, Rule (πR) selects an axiom Q from the theory P to discharge it. The selected axiom must *match* the goal, a notion that is captured by the second auxiliary judgment. Matching gives rise to a sequence $\mathcal{C}$ of new (and hopefully simpler) goals whose entailment is checked recursively.

The second auxiliary judgment $\Gamma; t : [Q] \models t' : \pi \rightsquigarrow \mathcal{C}$ focuses on the axiom Q and checks whether it matches the simple goal π. Again, there are three rules for the three possible forms the axiom can take. Rule (πL) expresses the base case where the axiom is identical to the goal and there are no new goals. Rule ($\Rightarrow$L) handles an implication axiom $Q_1 \Rightarrow Q_2$ by recursively checking whether Q_2 matches the goal. At the same time it yields a new goal Q_1 which needs to be entailed in order for the axiom to apply. Finally, Rule ($\forall$L) handles universal quantification by instantiating the quantified variable in a way that recursively yields a match.

Figure 7.3 Constraint Entailment: Tractable Specification

$\boxed{P; \Gamma \models t : Q}$ Constraint Entailment

$$\frac{P; \Gamma \models t : [Q]}{P; \Gamma \models t : Q}$$

$\boxed{P; \Gamma \models t : [Q]}$ Constraint Resolution

$$\frac{d \notin dom(P) \cup dom(\Gamma) \qquad \Gamma \vdash_{\text{CT}} Q_1 \rightsquigarrow v_1 \qquad P, d : Q_1; \Gamma \models t : [Q_2]}{P; \Gamma \models \lambda(d : v_1).\ t : [Q_1 \Rightarrow Q_2]}\ (\Rightarrow\text{R})$$

$$\frac{b \notin \Gamma \qquad P; \Gamma, b \models t : [Q]}{P; \Gamma \models \Lambda b.\ t : [\forall b.\ Q]}\ (\forall\text{R}) \qquad \frac{\begin{array}{c}(d : Q) \in P : \quad \Gamma; d : [Q] \models t : \pi \rightsquigarrow \mathcal{C} \\ \forall (d_i : Q_i) \in \mathcal{C} : \quad P; \Gamma \models t_i : [Q_i]\end{array}}{P; \Gamma \models [\overline{t_i}/\overline{d_i}]t : [\pi]}\ (\pi\text{R})$$

$\boxed{\Gamma; t : [Q] \models t' : \pi \rightsquigarrow \mathcal{C}}$ Constraint Matching

$$\frac{d \text{ fresh} \qquad \Gamma; t\ d : [Q_2] \models t' : \pi \rightsquigarrow \mathcal{C}}{\Gamma; t : [Q_1 \Rightarrow Q_2] \models t' : \pi \rightsquigarrow \mathcal{C}, d : Q_1}\ (\Rightarrow\text{L}) \qquad \frac{}{\Gamma; t : [\pi] \models t : \pi \rightsquigarrow \bullet}\ (\pi\text{L})$$

$$\frac{\Gamma; t\ v : [[\tau/b]Q] \models t' : \pi \rightsquigarrow \mathcal{C} \qquad \Gamma \vdash_{\text{TY}} \tau \rightsquigarrow v}{\Gamma; t : [\forall b.\ Q] \models t' : \pi \rightsquigarrow \mathcal{C}}\ (\forall\text{L})$$

It is not difficult to see that this type-directed formulation of entailment greatly reduces the number of proofs for given goal.[6] For instance, for the example above there is only one proof:

$$\frac{\dfrac{Eq\,a \in P \qquad \Gamma; [Eq\,a] \models Eq\,a \rightsquigarrow \bullet\ (\pi\text{L})}{P; \Gamma \models [Eq\,a]}\ (\pi\text{R})}{P; \Gamma \models Eq\,a}$$

Type-Directed Dictionary Construction The highlighted parts of Figure 7.3 present the specification of type-directed dictionary construction. The meaning of term t in the constraint entailment and the constraint resolution judgments

[6]Without loss of expressive power; the specification of Figure 7.3 does not commit to a particular choice any more than the specification of Figure 7.2.

is the expected: evidence term t is a proof that Q is satisfied. In the constraint matching relation, three kinds of evidence terms are involved: t is an evidence term witnessing given constraint Q, t' witnesses the wanted constraint π, and the residual constraints $\mathcal{C}$ are annotated with dictionary variables, representing the assumptions that remain to be proven.

Rules ($\Rightarrow$R) and ($\forall$R) are straightforward. Rule (πR) produces evidence term t by combining given proof d and residual evidence terms d_i. The complete proof for the wanted constraint π is obtained by substituting the actual evidence terms t_i for the temporary dictionary variables d_i, each witnessing a residual constraint Q_i.

Constraint matching operates by gradually specializing the evidence term t (witnessing constraint Q), until the given and the wanted constraints match. The base case is captured in Rule (πL), where the given and the wanted constraints are identical. In this case, the witness for both is the same. Rule ($\Rightarrow$L) focuses on the right-hand side of the implication (eliminates the arrow) by providing a fresh dictionary argument d to capture the residual constraint Q_1. Finally, the implicit specialization of the given constraint in Rule ($\forall$L) becomes explicit in the witness term, which is explicitly applied to the type it is instantiated with.

7.2.4 Remaining Non-determinism

While focusing makes the definition of constraint entailment type-directed, there are still two sources of non-determinism. As a consequence, the specification is still ambiguous and not an algorithm.

Overlapping Axioms The first source of non-determinism is that in Rule (πR) there may be multiple matching axioms that make the entailment go through. For applications of logic where proofs are irrelevant this is not a problem, but in Haskell where the proofs have computational content (namely the method implementations) this is a cause for concern. Haskell '98 also faces this problem. Consider two instances for the same type:

$$\begin{array}{l}\textbf{class}\ \mathit{Default}\ a\ \textbf{where}\ \{\ \mathit{default} :: a\ \}\\[1ex] \textbf{instance}\ \mathit{Default}\ \mathit{Bool}\ \textbf{where}\ \{\ \mathit{default} = \mathit{True}\ \}\\ \textbf{instance}\ \mathit{Default}\ \mathit{Bool}\ \textbf{where}\ \{\ \mathit{default} = \mathit{False}\ \}\end{array}$$

The two instances give rise to two different proofs for *Default Bool*, with distinct computational content (*True* vs. *False*). We steer away from this problem in the same way as Haskell '98, by requiring that instance declarations do not overlap. This does not rule out the possibility of distinct proofs for the same goal, but

Figure 7.4 Unambiguity

$\boxed{unamb(Q)}$ Unambiguity

$$\frac{\bullet \vdash_{\texttt{UNAMB}} Q}{unamb(Q)}\ \textsc{Unamb}$$

$\boxed{\overline{a} \vdash_{\texttt{UNAMB}} Q}$ Unambiguity

$$\frac{\overline{a} \subseteq fv(\pi)}{\overline{a} \vdash_{\texttt{UNAMB}} \pi}\ (\pi\text{U}) \qquad \frac{\overline{a}, a \vdash_{\texttt{UNAMB}} Q}{\overline{a} \vdash_{\texttt{UNAMB}} \forall a.\ Q}\ (\forall\text{U}) \qquad \frac{unamb(Q_1) \qquad \overline{a} \vdash_{\texttt{UNAMB}} Q_2}{\overline{a} \vdash_{\texttt{UNAMB}} Q_1 \Rightarrow Q_2}\ (\Rightarrow\text{U})$$

at least distinct proofs have the same computational content. Consider a class hierarchy where C is the superclass of both D and E.

$$\begin{array}{l} \textbf{class}\ C\ a\ \textbf{where}\ \{\ \ldots\ \} \\ \textbf{class}\ C\ a \Rightarrow D\ a\ \textbf{where}\ \{\ \ldots\ \} \\ \textbf{class}\ C\ a \Rightarrow E\ a\ \textbf{where}\ \{\ \ldots\ \} \end{array}$$

This gives rise to the superclass axioms $\forall a.\ D\ a \Rightarrow C\ a$ and $\forall a.\ E\ a \Rightarrow C\ a$. Given additionally two local constraints $D\ \tau$ and $E\ \tau$, we have two ways to establish $C\ \tau$. The proofs are distinct, yet ultimately the computational content is the same. This is easy to see as only instances supply the computational content and there can be at most one instance for any given type τ.

In summary, non-overlap of instances is sufficient to ensure *coherence*.

Guessing Polymorphic Instantiation A second source of ambiguity is that Rule ($\forall$L) requires guessing an appropriate type τ for substituting the type variable b. Guessing is problematic because there is an infinite number of types to choose from and more than one of those choices can make the entailment work out. Choosing an appropriate type is a problem for the type inference algorithm in the next section. As we discussed in Section 6.5, different choices leading to different proofs is a more fundamental problem that also manifests itself in Haskell'98.

Indeed, Definition 4 captures the requirement that all quantified type variables appear in the head of an axiom. Because our axioms have a more general, recursively nested form, we generalize this requirement in a recursively nested fashion. The predicate $unamb(Q)$ in Figure 7.4 formalizes the requirement in terms of the auxiliary judgment $\overline{a} \vdash_{\texttt{UNAMB}} Q$, where $\overline{a}$ are type variables that need to be determined by the head of Q. Rule (πU) constitutes the base case where π

is the head and contains the determinable type variables $\overline{a}$. Rule ($\forall$U) processes a quantifier by adding the new type variable to the list of determinable type variables $\overline{a}$. Finally, Rule ($\Rightarrow$U) checks whether the head Q_2 of the implication determines the type variables $\overline{a}$. It also recursively checks whether Q_1 is unambiguous on its own. The latter check is necessary because left-hand sides of implications are themselves added as axioms to the theory in Rule ($\Rightarrow$R); hence they must be well-behaved on their own.

The predicate $\mathit{unamb}(Q)$ must be imposed on all constraints that are added to the theory. This happens in four places: the instance axioms added in Rule Instance, the superclass axioms added in Rule Class, the local axioms added when checking against a given signature in Rule ($\Rightarrow$I) and the local axioms added during constraint entailment checking in Rule ($\Rightarrow$R). These four places can be traced back to three places in the syntax: class and instance heads, and (method) signatures.

7.3 Type Inference with Elaboration

We now turn to type inference and elaboration in the presence of quantified class constraints. Similarly to the declarative specification, the extension of the system with quantified constraints affects the algorithm in a minimal way: the relevant relations and functions for terms ($\Gamma \vdash_{\mathtt{TM}} e : \tau \rightsquigarrow t \mid \mathcal{C} \ ; E$), types ($\mathit{elab}_{\text{TY}}(\sigma) = \upsilon$), declarations ($\Gamma \vdash_{\mathtt{CLS}} \mathit{cls} : P; \ \Gamma' \rightsquigarrow \overline{\mathit{decl}}$ (class declarations), $P; \Gamma \vdash_{\mathtt{INS}} \mathit{ins} : P' \rightsquigarrow \mathit{decl}$ (class instances), and $P; \Gamma \vdash_{\mathtt{VAL}} \mathit{val} : \Gamma' \rightsquigarrow \mathit{decl}$ (value bindings)), and programs ($\vdash_{\mathtt{PGM}} \mathit{pgm} : P; \Gamma \rightsquigarrow \overline{\mathit{decl}}$) are identical to the ones for the basic system (Section 6.4).

The relations and functions that are affected by the more general form of constraints are constraint elaboration ($\mathit{elab}_{\text{CT}}(Q) = \upsilon$) and, notably, constraint entailment with dictionary construction ($\overline{a}; \mathcal{A} \models \mathcal{C}_1 \rightsquigarrow \mathcal{C}_2; \eta$). These relations are the subject of the remainder of this section.

7.3.1 Constraint Elaboration

Elaboration of constraints takes the form $\mathit{elab}_{\text{CT}}(Q) = \upsilon$ and conservatively extends the definition we gave in Section 6.4, to account for the new syntactic forms. It is given by the following clauses:

$$\begin{aligned}
\mathit{elab}_{\text{CT}}(\mathtt{TC}\ \tau) &= T_{\mathtt{TC}}\ \mathit{elab}_{\text{TY}}(\tau) \\
\mathit{elab}_{\text{CT}}(\forall a.\ Q) &= \forall a.\ \mathit{elab}_{\text{CT}}(Q) \\
\mathit{elab}_{\text{CT}}(Q_1 \Rightarrow Q_2) &= \mathit{elab}_{\text{CT}}(Q_1) \rightarrow \mathit{elab}_{\text{CT}}(Q_2)
\end{aligned}$$

Similarly to the basic system, elaboration of constraints follows directly from the specification of well-formedness and elaboration of constraints; function $elab_{\text{CT}}(Q) = v$ essentially implements the elaboration-part of specification $\Gamma \vdash_{\text{CT}} Q \rightsquigarrow v$ (see Section 7.2). For example, the constraint corresponding to the *Show* instance for type *HPerf* of Section 7.1.2

$$\forall f.\ \forall a.\ Show\ a \Rightarrow (\forall x.\ Show\ x \Rightarrow Show\ (f\ x)) \Rightarrow Show\ (HPerf\ f\ a)$$

is elaborated into the type

$$\forall f.\ \forall a.\ T_{Show}\ a \rightarrow (\forall x.\ T_{Show}\ x \rightarrow T_{Show}\ (f\ x)) \rightarrow T_{Show}\ (HPerf\ f\ a)$$

7.3.2 Constraint Solving

Similarly to the basic system, the type class and equality constraints derived from terms are solved with two algorithms: unification and class constraint entailment. Unification is identical to that of the basic system (function $unify(\overline{a}; E) = \theta_{\perp}$, Section 6.4), so we only discuss the constraint entailment algorithm here.

Solving Type Class Constraints Figure 7.5 defines the judgment for solving type class constraints; it takes the form $\overline{a};\ P \models \mathcal{C}_1 \rightsquigarrow \mathcal{C}_2;\ \eta$. Given a set of untouchable type variables $\overline{a}$ and a theory P, it (exhaustively) replaces a set of constraints $\mathcal{C}_1$ with a set of *simpler*, residual constraints $\mathcal{C}_2$, via the auxiliary judgment $\overline{a};\ P \models [\mathcal{Q}] \rightsquigarrow \mathcal{C};\ \eta$, explained below.

Simplification Auxiliary judgment $\overline{a};\ P \models [\mathcal{Q}] \rightsquigarrow \mathcal{C};\ \eta$ uses the theory P to simplify a single constraint $\mathcal{Q}$ to a set of simpler constraints without instantiating any of the untouchable type variables $\overline{a}$. Following the focusing approach, the judgment is defined by three rules, one for each of the syntactic forms of the goal $\mathcal{Q}$.

Rules ($\Rightarrow$R) and ($\forall$R) recursively simplify the head of the goal. Observe that we add the bound variable b to the untouchables $\overline{a}$ when going under a binder in Rule ($\forall$R). Once the goal is stripped down to a simple class constraint Π, Rule (πR) selects an axiom $\mathcal{Q}$ whose head matches the goal, and uses it to replace the goal with a set of simpler constraints $\mathcal{C}$ (a process known as *context reduction* (Jones, 1995b)). Goal matching is performed by judgment $\overline{a}; [\mathcal{Q}] \models \Pi \rightsquigarrow \mathcal{C};\ \theta;\ \eta$, discussed below.

Matching Auxiliary judgment $\overline{a}; [\mathcal{Q}] \models \Pi \rightsquigarrow \mathcal{C};\ \theta;\ \eta$ focuses on the axiom $\mathcal{Q}$ and checks whether it matches the simple goal Π. The main difference between

Figure 7.5 Constraint Entailment with Dictionary Construction

$\boxed{\overline{a};\ P \models \mathcal{C}_1 \rightsquigarrow \mathcal{C}_2; \eta}$ Constraint Solving Algorithm

$$\frac{\nexists \mathcal{Q} \in \mathcal{C}_1 : \quad \overline{a};\ P \models [\mathcal{Q}] \rightsquigarrow \mathcal{C}_2 ; \eta}{\overline{a};\ P \models \mathcal{C}_1 \rightsquigarrow \mathcal{C}_1; \bullet}\ \textsc{Stop}$$

$$\frac{\overline{a};\ P \models [\mathcal{Q}] \rightsquigarrow \mathcal{C}_2 ; \eta_1 \qquad \overline{a};\ P \models \mathcal{C}_1, \mathcal{C}_2 \rightsquigarrow \mathcal{C}_3; \eta_2}{\overline{a};\ P \models \mathcal{C}_1, \mathcal{Q} \rightsquigarrow \mathcal{C}_3; (\eta_2 \cdot \eta_1)}\ \textsc{Step}$$

$\boxed{\overline{a};\ P \models [\mathcal{Q}] \rightsquigarrow \mathcal{C} ; \eta}$ Constraint Simplification

$$\frac{\begin{array}{c}\overline{a};\ P,_{\text{L}} (d_1 : Q_1) \models [d_2 : Q_2] \rightsquigarrow \overline{(d : Q)} ; \eta \\ \overline{d'}, d_1, d_2 \text{ fresh} \qquad \eta' = [\lambda(d_1 : \mathit{elab}_{\text{CT}}(Q_1)).\ [\overline{d'\ d_1 / d}](\eta(d_2))/d_0]\end{array}}{\overline{a};\ P \models [d_0 : Q_1 \Rightarrow Q_2] \rightsquigarrow \overline{(d' : Q_1 \Rightarrow Q)} ; \eta'}\ (\Rightarrow\text{R})$$

$$\frac{\overline{d'}, d_Q \text{ fresh} \qquad \overline{a}, b;\ P \models [d_Q : Q_0] \rightsquigarrow \overline{(d : Q)} ; \eta}{\overline{a};\ P \models [d_0 : \forall b.\ Q_0] \rightsquigarrow \overline{(d' : \forall b.\ Q)} ; [\Lambda b.\ [\overline{d'\ b / d}](\eta(d_Q))/d_0]}\ (\forall\text{R})$$

$$\frac{\mathcal{Q} \in P : \quad \overline{a}; [\mathcal{Q}] \models \Pi \rightsquigarrow \mathcal{C};\ \theta ; \eta}{\overline{a};\ P \models [\Pi] \rightsquigarrow \mathcal{C} ; \eta}\ (\pi\text{R})$$

$\boxed{\overline{a}; [\mathcal{Q}] \models \Pi \rightsquigarrow \mathcal{C};\ \theta ; \eta}$ Constraint Matching

$$\frac{d_1, d_2 \text{ fresh} \qquad \overline{a}; [d_2 : Q_2] \models \Pi \rightsquigarrow \mathcal{C};\ \theta ; \eta}{\overline{a}; [d : Q_1 \Rightarrow Q_2] \models \Pi \rightsquigarrow \mathcal{C}, d_1 : \theta(Q_1);\ \theta ; [d\ d_1/d_2] \cdot \eta}\ (\Rightarrow\text{L})$$

$$\frac{d' \text{ fresh} \qquad \overline{a}; [d' : Q] \models \Pi \rightsquigarrow \mathcal{C};\ \theta ; \eta}{\overline{a}; [d : \forall b.\ Q] \models \Pi \rightsquigarrow \mathcal{C};\ \theta ; [d\ (\theta(b))/d'] \cdot \eta}\ (\forall\text{L})$$

$$\frac{\theta = \mathit{unify}(\overline{a}; \tau_1 \sim \tau_2)}{\overline{a}; [d' : \texttt{TC}\ \tau_1] \models d : \texttt{TC}\ \tau_2 \rightsquigarrow \bullet;\ \theta ; [d'/d]}\ (\pi\text{L})$$

this algorithmic relation and its declarative specification in Figure 7.3 lies in the type substitution θ. Instead of guessing a type for instantiating a polymorphic axiom in Rule ($\forall$L) (top-down), we defer the choice until the head of the axiom is met, in Rule (πL) (bottom-up). Observe that Rule ($\forall$L) does not record b as untouchable, effectively turning it into a unification variable. Thus, by unifying the head of the axiom with the goal we can determine without guessing an instantiation for all top-level quantifiers, captured by the type substitution θ.

Search As Section 7.2.4 has remarked, there may be multiple matching axioms, e.g., due to overlapping superclass axioms. The straightforward algorithmic approach to the involved non-determinism is search, possibly implemented by backtracking. The GHC Haskell implementation can employ a heuristic to keep this search shallow. It does so by using the superclass constraints very selectively: whenever a new local constraint is added to the theory, it pro-actively derives all its superclasses and adds them as additional local axioms (just as function $\textsc{ScClosure}(\overline{a}, P) = (\mathcal{A}, \mathbb{E})$ does for the basic system). When looking for a match, it does not consider the superclass axioms and prefers the local axioms over the instance axioms. If a matching local axiom exists, it immediately discharges the entire goal without further recursive resolution. This is the case because in regular Haskell local axioms are always simple class constraints π.

In our setting, we can also implement (a modified version of) GHC's heuristic, but this does not obviate the need for deep search. The reason is that our local axioms are not necessarily simple axioms, and matching against them may leave residual goals that require further recursive resolution. When that recursive resolution gets stuck, we have to backtrack over the choice of axiom. Consider the following example.

$$\begin{array}{l}\textbf{class}\ (E\ a \Rightarrow C\ a) \Rightarrow D\ a\\ \textbf{class}\ (G\ a \Rightarrow C\ a) \Rightarrow F\ a\end{array}$$

Given local axioms $D\ a$, $F\ a$ and $G\ a$, consider what happens when we resolve the goal $C\ a$. The superclasses $E\ a \Rightarrow C\ a$ and $G\ a \Rightarrow C\ a$ of respectively $D\ a$ and $F\ a$ both match this goal. If we pick the first one, we get stuck when recursively resolving $E\ a$. However, if we backtrack and consider the second one instead, we can recursively resolve $G\ a$ against the given local constraint.

In summary, because we do not see a general way to avoid search, our prototype implementation uses backtracking for choosing between the different axioms.[7]

[7]It is worth mentioning that the rules of Figure 7.5 conservatively extend standard Haskell resolution, both in terms of expressivity and performance.

7.3.3 Dictionary Construction

The entailment algorithm of Figure 7.5 constructs explicit witness proofs (in the form of dictionary substitutions) while entailing a constraint.

Simplification The evidence substitution η in the simplification relation shows how to construct a witness for the wanted constraint $\mathcal{Q}$ from the simpler constraints $\mathcal{C}$ and program theory P.

The goal of Rule ($\Rightarrow$R) is to build an evidence substitution η', which constructs a proof for $(d_0 : Q_1 \Rightarrow Q_2)$ from the proofs $\overline{d'}$ for the simpler constraints $\overline{Q_1 \Rightarrow Q}$. It is instructive to consider the generated evidence substitution in parts, also taking the types into account:

1. η illustrates how to generate a proof for $(d_2 : Q_2)$, from the local assumption $(d_1 : Q_1)$ and local residual constraints $(\overline{d : Q})$.
2. $[\overline{d'\ d_1/d}]$ generates proofs for the (local) residual constraints $(\overline{d : Q})$, by applying the residual constraints $(\overline{d' : Q_1 \Rightarrow Q})$ to the local assumption $(d_1 : Q_1)$.
3. $([\overline{d'\ d_1/d}] \cdot \eta)(d_2)$ is a proof for Q_2, under assumptions $(d_1 : Q_1)$ and $(\overline{d' : Q_1 \Rightarrow Q})$.
4. Finally, we construct the proof for $(d_0 : Q_1 \Rightarrow Q_2)$ by explicitly abstracting over d_1: $\lambda(d_1 : \upsilon_1).\ [\overline{d'\ d_1/d}](\eta(d_2))$

Rule ($\forall$R) proceeds similarly. Finally, Rule (πR) generates the evidence substitution via constraint matching, which we discuss next.

Matching Similarly, the evidence substitution η in the matching relation shows how to construct a witness for the wanted constraint Π from the simpler constraints $\mathcal{C}$ and program theory P.

Rule ($\Rightarrow$L) generates two fresh dictionary variables, d_1 for the residual constraint $\theta(Q_1)$, and d_2 for the local assumption Q_2. Finally, dictionary d_2 is replaced by the application of the dictionary transformer d to the residual dictionary d_1. Rule ($\forall$L) behaves similarly. The instantiation of the axiom d becomes explicit, by applying it to the chosen type $\theta(b)$. Finally, Rule (πL) is straightforward: since the wanted and the given constraints are identical (given that they unify), the wanted dictionary d is replaced by the given d'.

Figure 7.6 Termination Condition

$\boxed{head(Q) = \pi}$ Constraint Head

$$\begin{array}{lcl} head(\pi) & = & \pi \\ head(\forall a.\ Q) & = & head(Q) \\ head(Q_1 \Rightarrow Q_2) & = & head(Q_2) \end{array}$$

$\boxed{term(Q)}$ Termination Condition

$$\frac{}{term(\pi)}\ (\pi\mathrm{T}) \qquad \frac{term(Q)}{term(\forall a.\ Q)}\ (\forall\mathrm{T})$$

$$\frac{\begin{array}{c} term(Q_1) \quad term(Q_2) \quad \pi_1 = head(Q_1) \quad \pi_2 = head(Q_2) \\ \|\pi_1\| < \|\pi_2\| \quad \forall a \in fv(Q_1) \cup fv(Q_2) : \quad occ_a(\pi_1) \leq occ_a(\pi_2) \end{array}}{term(Q_1 \Rightarrow Q_2)}\ (\Rightarrow\mathrm{T})$$

7.4 Termination of Resolution

Termination of resolution is the cornerstone of the overall termination of type inference. This section discusses how to enforce termination by means of syntactic conditions on the axioms. These conditions are adapted from those of Cochis (Schrijvers et al., 2017) and generalize the earlier conditions for Haskell by Sulzmann et al. (2007b).

Overall Strategy We show termination by characterizing the resolution process as a (resolution) tree with goals in the nodes and axioms on the (multi-)edges. The initial goal sits at the root of the tree. A multi-edge from a parent node to its children presents an axiom that matches the parent node's goal and its children are the residual goals. Resolution terminates iff the tree is finite. Hence, if it does not terminate, there is an infinite path from the root in the tree, that denotes an infinite sequence of axiom applications.

To show that there cannot be such an infinite path, we use a norm $\|\cdot\|$ that maps the head of every goal Q to a natural number, its size. The size of a class constraint `TC` τ is the size of its type parameter τ, which is identical to the definition we gave for the basic system (Section 6.5). The head of a constraint is computed by function $head(Q)$, presented in Figure 7.6 (top).

If we can show that this size strictly decreases from any parent goal to its children, then we know that, because the order on the natural numbers is well-founded, on any path from the root there is eventually a goal that has no children.

Termination Condition It is trivial to show that the size strictly decreases, if we require that every axiom makes it so. This requirement is formalized as the termination condition of axioms $term(Q)$, presented in Figure 7.6 (bottom).

Rule ($\Rightarrow$T) for $Q_1 \Rightarrow Q_2$ enforces the main condition, that the size of the residual constraint's head π_1 is strictly smaller than the head π_2 of Q_2. In addition, the rule ensures that this property is stable under type substitution. Consider for instance the axiom $\forall a.\ C\ (a \to a) \Rightarrow C\ (a \to Int \to Int)$. The head's size 5 is strictly greater than the context constraint's size 3. Yet, if we instantiate a to $(Int \to Int \to Int)$, then the head's size becomes 10 while the context constraint's size becomes 11. Declaratively, we can formulate stability as:

$$\forall \theta. dom(\theta) \subseteq fv(Q_1) \cup fv(Q_2) \Rightarrow \|\theta(\pi_1)\| < \|\theta(\pi_2)\|$$

The rule uses instead an equivalent algorithmic formulation which states that the number of occurrences of any free type variable a may not be larger in π_1 than in π_2. Here the number of occurrences of a type variable a in a class constraint $\texttt{TC}\ \tau$ (denoted as $occ_a(\texttt{TC}\ \tau)$) is the same as the number of free occurrences of a in the parameter τ, where function $occ_a(\tau)$ is defined as for the basic system (Section 6.5).

Finally, as the constraints have a recursive structure whereby their components are themselves used as axioms, the rules also enforce the termination condition recursively on the components.

Superclass Condition If we could impose the termination condition above on all axioms in the theory P, we would be set. Unfortunately, this condition is too strong for the superclass axioms. Consider the superclass axiom $\forall a.\ Ord\ a \Rightarrow Eq\ a$ of the standard Haskell '98 Ord type class. Here both $Ord\ a$ and $Eq\ a$ have size 1; in other words, the size does not strictly decrease and so the axiom does not satisfy the termination condition.

To accommodate this and other examples, we impose an alternative condition for superclass axioms. This superclass condition relaxes the strict size decrease to a non-strict size decrease and makes up for it by requiring that the superclass relation forms a *directed acyclic graph* (DAG). The superclass relation is defined on type classes as follows:

Definition 5 (Superclass Relation)**.** *Given a class declaration*

$$\textbf{class}\ (Q_1, \ldots, Q_n) \Rightarrow \texttt{TC}\ a\ \textbf{where}\ \{\ f :: \sigma\ \}$$

each type class TC_i *is a superclass of* TC*, where* $\mathit{head}(Q_i) = \mathit{TC}_i\ \tau_i$*.*

Observe that the DAG induces a well-founded partial order on type classes. Hence, on any path in the resolution tree, any uninterrupted sequence of superclass axiom applications has to be finite. For the length of such a sequence, the size of the goal does not increase (but might not decrease either). Yet, after a finite number of steps the sequence has to come to an end. If the path still goes on at that point, it must be due to the application of an instance or local axiom, which strictly decreases the goal size. Hence, overall we have preserved the variant that the goal size decreases after a bounded number[8] of steps.

Other Properties Finally, although we have not proven it formally yet, we are confident that soundness of type inference and preservation of typing under elaboration hold independently of termination (and thus are not affected by whether the termination conditions are met). Such a property is crucial for integrating our algorithm within GHC in the future, where flags such as `UndecidableInstances` are heavily used. Similarly, elaboration is coherent if overlapping instances are disallowed; as we discuss in Section 10.3.2, the formal proof of all these properties constitutes part of our future work.

7.5 Related Work

This section discusses related work, focusing mostly on comparing our approach with existing encodings/workarounds in Haskell. The history of quantified class constraints and their demand in previous research was already discussed in Section 7.1.1.

The Coq **Proof Assistant** Coq provides very flexible support for type classes (Sozeau and Oury, 2008) and allows for arbitrary formulas in class and instance contexts—actually the contexts are just parameters. For instance, we can model the *Trans* class as:

```
Class Trans (T : (Type -> Type) -> Type -> Type)
  ‘{forall M, ‘{Monad M} -> Monad (T M)} :=
  { lift : forall A M, ‘{Monad M} -> M A -> (T M) A }.
```

[8]bounded by the height of the superclass DAG

The downside of Coq's flexibility is that resolution can be ambiguous and non-terminating. The accepted workaround is for the programmer to perform resolution manually when necessary. This is acceptable in the context of Coq's interactive approach to proving, but would mean a great departure from Haskell's non-interactive type inference.

Trifonov's Workaround and Monatron Trifonov (2003) gives an encoding of quantified class constraints in terms of regular class constraints. The encoding introduces a new type class that encapsulates the quantified constraint, e.g. $Monad\_t\ t$ for $\forall m.\ Monad\ m \Rightarrow Monad\ (t\ m)$, and that provides the implied methods under a new name. This expresses the *Trans* problem as follows:

```
class Monad_t t where
  treturn :: Monad m ⇒ a → t m a
  tbind   :: Monad m ⇒ t m a → (a → t m b) → t m b

class Monad_t t ⇒ Trans t where
  lift :: Monad m ⇒ m a → t m a
```

While this approach captures the intention of the quantified constraint, it does not enable the type checker to see that $Monad\ (t\ m)$ holds for any transformer t and monad m. While the monad methods are available for $t\ m$, they do not have the usual name.

For this reason, Trifonov presents a further (non-Haskell '98) refinement of the encoding, which was adopted by the Monatron (Jaskelioff, 2011) library[9] among others. A non-essential difference is that Monatron merges the above *Monad_t* and *Trans* into a single class:

```
class MonadT t where
  lift    :: Monad m ⇒ m a → t m a
  treturn :: Monad m ⇒ a → t m a
  tbind   :: Monad m ⇒ t m a → (a → t m b) → t m b
```

The key novelty is that it also makes the methods *treturn* and *tbind* available under their usual name with a single *Monad* instance for all monad transformers.

```
instance (Monad m, MonadT t) ⇒ Monad (t m) where
  return = treturn
  (>>=)  = tbind
```

With these definitions the monad transformer composition does type check. Unfortunately, the head of the $Monad\ (t\ m)$ instance is highly generic and easily overlaps with other instances.

[9]For the implementation see `https://hackage.haskell.org/package/Monatron`.

The MonadZipper Because they found Monatron's overlapping instances untenable, Schrijvers and Oliveira (2011) presented a different workaround for this problem in the context of their monad zipper datatype, which is an extended form of transformer composition. Their solution adds a method *mw* to the *Trans* type class:

```
class Trans t where
  lift :: Monad m ⇒ m a → t m a
  mw :: Monad m ⇒ MonadWitness t m
```

For any monad m this method returns a GADT (Peyton Jones et al., 2006) witness for the fact that $t\ m$ is a monad. This is possible because with GADTs, type class instances can be stored in the data constructors.

```
data MonadWitness (t :: (∗ → ∗) → (∗ → ∗)) m where
  MW :: Monad (t m) ⇒ MonadWitness t m
```

By pattern matching on the witness of the appropriate type the programmer can bring the required *Monad* $(t_2\ m)$ constraint into scope to satisfy the type checker.

```
instance (Trans t₁, Trans t₂) ⇒ Trans (t₁ ∗ t₂) where
  lift :: ∀m. ∀a. Monad m ⇒ m a → (t₁ ∗ t₂) m a
  lift = case (mw :: MonadWitness t₂ m) of
           MW → C · lift · lift

  mw = ...
```

The downside of this approach is that it offloads part of the type checker's work on the programmer. As a consequence, the code becomes cluttered with witness manipulation.

The constraint Library Kmett's `constraint` library (Kmett, 2017) provides generic infrastructure for reifying quantified constraints in terms of GADTs, generalizing the MonadZipper solution above. Additionally, it complements the encoding with ample utilities for the manipulation of such constraints. Unfortunately, it suffers from the same problem: passing, construction and deconstruction of dictionaries needs to be manually performed by the programmer.

Corecursive Resolution Fu et al. (2016) address the divergence problem that arises for generic nested datatypes. They turn the diverging resolution with user-supplied instances into a terminating resolution in terms of automatically derived instances. These auxiliary instances are derived specifically to deal with the query at hand; they shift the pattern of divergence to the term-level in the form of co-recursively defined dictionaries. The authors do point out that the class of divergent cases they support is limited and that deriving quantified instances would be beneficial.

Cochis The calculus of coherent implicits, COCHIS (Schrijvers et al., 2017), and its predecessor, the *implicit calculus* (Oliveira et al., 2012), have been a major inspiration for our work. Just like our calculus, COCHIS supports recursive resolution of quantified constraints using a focusing-based algorithm. Yet, there are a number of significant differences. Firstly, COCHIS does not feature a separate syntactic sort for type classes, but implicitly resolves regular terms in the Scala tradition. As a consequence, it does not distinguish between instance and superclass axioms, e.g., for the sake of enforcing termination and coherence. Perhaps more significantly, COCHIS features local "instances" as opposed to our globally scoped instances. Local instances may overlap with one another and coherence is obtained by prioritizing those instances that are introduced in the innermost scope. This way COCHIS's resolution is entirely deterministic, while ours is non- deterministic (yet coherent) due to overlapping local and superclass axioms.

7.6 Scientific Output

This chapter has presented a fully fledged design of quantified class constraints, a feature that has been requested by users for eighteen years. We have shown that this feature significantly increases the modelling power of type classes, while at the same enables a terminating type class resolution for a larger class of applications.

Most of the material found in this chapter is drawn from the following publication:

> Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira and Philip Wadler (2017). Quantified Class Constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell '17, pp. 148–161, Oxford, UK, September 7–8, 2017.

In addition to the material presented in the above publication, this chapter also presents the specification of elaboration (Section 7.2.3); this allows for the formal statement of the correctness of dictionary construction of Section 7.3.

Furthermore, we have implemented the type inference and elaboration algorithm of Section 7.3 in a prototype compiler, which is available at

`https://github.com/gkaracha/quantcs-impl`

The prototype incorporates higher-kinded datatypes and performs type inference, elaboration into System F, and type checking of the generated code.

Within this multi-author work, the contribution of each author has been the following:

- The key idea of using the focusing technique has been introduced by Schrijvers et al. (2017).
- The specification of typing and elaboration, as well as the type inference algorithm with evidence generation has been collaboratively designed by the author of this thesis, Gert-Jan Bottu, and Tom Schrijvers.
- The initial prototype implementation of quantified class constraints has been collaboratively developed by the author of this thesis and Gert-Jan Bottu. The latest version is a revision by the author of this thesis.

Chapter 8

Functional Dependencies

In this chapter we present the second of our extensions to type classes: *Functional Dependencies.*

The idea of functional dependencies has been introduced eighteen years ago (Jones, 2000) and they have enjoyed widespread use since their introduction in Haskell compilers. Nevertheless, their design and implementation in GHC has never been completely faithful to their specification. In this chapter, we revisit the idea of functional dependencies and give a novel specification of their semantics, which allows their proper integration within the Haskell eco-system.

More specifically, the chapter is structured as follows: Section 8.1 motivates the development of functional dependencies and presents infelicities of their existing formalizations and implementations. Section 8.2 gives a logical interpretation of functional dependencies in terms of first-order logic; Section 8.3 utilizes this interpretation to give our specification of typing for type classes with functional dependencies. In order to capture the inherently non-parametric semantics of functional dependencies, Section 8.4 presents System F_C (Sulzmann et al., 2007a), an extension of System F with open, non-parametric type functions; System F_C serves as our target language. Section 8.5 develops a type-inference algorithm for type classes with functional dependencies, which elaborates source programs to System F_C. Section 8.6 formulates the most important meta-theoretical properties of our development. Section 8.7 discusses related work. Finally, Section 8.8 concludes and summarizes the scientific output of this work.

8.1 Motivation

This section motivates the development of functional dependencies that is presented in the remainder of this chapter. Since functional dependencies made their first appearance eighteen years ago (Jones, 2000), Section 8.1.1 gives a brief historical note on their development since they were originally introduced in Haskell. Next, Section 8.1.2 informally presents functional dependencies through a series of examples. Sections 8.1.3, 8.1.4, and 8.1.5 discuss the three main challenges we aim to address. Finally, Section 8.1.6 discusses our approach in intuitive terms; our formalization is presented in subsequent sections.

8.1.1 A History of Functional Dependencies

After type classes were introduced by Wadler and Blott (1989), the feature was quickly and naturally generalized from single-parameter predicates over types to relations over multiple types. Unfortunately, these so-called *multi-parameter* type classes easily give rise to ambiguous situations where the combination of types in the relation can, as a matter of principle, not be uniquely determined. In many situations a functional relation between the types that inhabit a multi-parameter type class is intended. Hence, Jones proposed the *functional dependency* language extension (Jones, 2000), which specifies that one class parameter determines another.

Functional dependencies became quite popular, not only to resolve ambiguity, but also as a device for type-level computation, which was used to good effect, e.g., for operations on heterogeneous collections (Kiselyov et al., 2004). They were supported by Hugs, Mercury, Habit (Jones, 2010) and also GHC. However, the implementation in GHC has turned out to be problematic: as far as we know, it is not possible to elaborate all well-typed programs with functional dependencies into GHC's original typed intermediate language based on System F (Girard, 1972; Reynolds, 1974, 1983). As a consequence, GHC rejects programs that are perfectly valid according to the theory of Sulzmann et al. (2007b). What's more, GHC's type checker does accept programs that violate the functional dependency property.

With the advent of associated types (Chakravarty et al., 2005a) (a.k.a. type families) came a new means for type-level computation, with a functional notation. Because it too cannot be elaborated into System F, a new extended core calculus with type-equality coercions was developed, called System F_C (Sulzmann et al., 2007a). However, it was never investigated whether functional dependencies would benefit from this more expressive core language. To date functional dependencies remain a widely popular, yet unreliably implemented feature. They are even gaining new relevance as functional dependency annotations on type families are being investigated (Stolarek et al., 2015).

Furthermore, as Jones and Diatchki (2008) rightly pointed out, the interaction of functional dependencies with other features has not been formally studied. In fact, recent discussions in the Haskell community indicate an interest in the interaction of functional dependencies with type families (GHC feature request #11534). Moreover, the unresolved nature of the problem has ramifications beyond Haskell, as PureScript has also recently adopted functional dependencies.[1]

8.1.2 Functional Dependencies

The concept of a *functional dependency* originates in relational database theory (Silberschatz et al., 2006): a relation R satisfies the functional dependency $X \rightarrow Y$, where X and Y are attributes of R, iff:

$$\forall (x, y_1), (x, y_2) \in R.\ y_1 = y_2 \tag{8.1}$$

In other words, every X value in R is associated with precisely one Y value. The feature was first introduced in Haskell by Jones (2000) as an extension to multi-parameter type classes and has been widely used over the years. The following variant of the well-known *collection* example (Peyton Jones, 1997) illustrates the feature:

$$\begin{array}{l}\textbf{class}\ \mathit{Coll}\ c\ e \mid c \rightarrow e\ \textbf{where}\\ \quad \mathit{singleton} :: e \rightarrow c\end{array}$$

The class *Coll* abstracts over collection types c with element type e. The functional dependency ($c \rightarrow e$) expresses that *"c uniquely determines e"*. Hence, functional dependencies have exactly the same meaning in Haskell as in relational database theory. After all, a multi-parameter type class like *Coll* can easily be seen as a relation over types. There is one main difference between Haskell type classes and database relations: the latter are typically defined extensionally (i.e., as a finite enumeration of tuples). In contrast, the former are given intensionally by means of type class instances (which can be seen as Horn clauses) from which infinitely many tuples can be derived by means of type class resolution.

Besides supporting functional dependencies syntactically as documentation for the programmer, Haskell also supports functional dependencies semantically in two ways. Firstly, it enforces that the type class instances respect the functional

[1]`http://goo.gl/V55whi`

dependency. This means for example that we cannot define two instances that associate different element types with the same collection type:

$$\begin{array}{l}\textbf{instance } \mathit{Coll\ Integer\ Bit}\ \ \textbf{where } \{\mathit{singleton}\ c = \ldots\} \\ \textbf{instance } \mathit{Coll\ Integer\ Byte}\ \textbf{where } \{\mathit{singleton}\ c = \ldots\}\end{array}$$

Secondly, functional dependencies give rise to more precise types and resolve ambiguities. For example, ignoring the functional dependency of *Coll*, function:

$$\mathit{singleton}_2\ c = \mathit{singleton}\ (\mathit{singleton}\ c)$$

has the ambiguous type:

$$\mathit{singleton}_2 :: (\mathit{Coll}\ c_1\ e, \mathit{Coll}\ c_2\ c_1) \Rightarrow e \rightarrow c_2$$

Type variable c_1 does not appear on the right of the $\Rightarrow$, which in turn means that no matter what argument we call $\mathit{singleton}_2$ on, c_1 will not be determined. Such ambiguous programs are typically rejected, since their run-time behavior is unpredictable (Section 8.6).

Yet, the functional dependency expresses that c_1 is not free, but uniquely determined by the choice of c_2, which will be fixed at call sites. Hence, if we take the functional dependency into account, $\mathit{singleton}_2$'s type is no longer ambiguous.

While functional dependencies are well-understood in the world of databases (Silberschatz et al., 2006), their incarnation in Haskell is still surrounded by a number of major algorithmic challenges and open questions.

8.1.3 Challenge 1: Enforcing Functional Dependencies

Unfortunately, the current implementation of functional dependencies in the Glasgow Haskell Compiler does not enforce the functional dependency property (Equation 8.1) in all circumstances.[2] The reason is that no criteria have been identified to do so under the *Liberal Coverage Condition* (Sulzmann et al., 2007b, Def. 15), which regulates ways of defining functional dependencies indirectly

[2]See for example GHC bug reports #9210 and #10675.

through instance contexts. The following example illustrates the problem.

$$\begin{array}{l}
\textbf{class}\ C\ a\ b\ c \mid a \to b\ \textbf{where}\ \{\mathit{foo} :: a \to c \to b\} \\[4pt]
\textbf{class}\ D_1\ a\ b \mid a \to b\ \textbf{where}\ \{\mathit{bar} :: a \to b\} \\
\textbf{class}\ D_2\ a\ b \mid a \to b\ \textbf{where}\ \{\mathit{baz} :: a \to b\} \\[4pt]
\textbf{instance}\ D_1\ a\ b \Rightarrow C\ [a]\ [b]\ \mathit{Int}\quad \textbf{where}\ \{\mathit{foo}\ [a]\ \_ = [\mathit{bar}\ a]\} \\
\textbf{instance}\ D_2\ a\ b \Rightarrow C\ [a]\ [b]\ \mathit{Bool}\ \textbf{where}\ \{\mathit{foo}\ [a]\ \_ = [\mathit{baz}\ a]\} \\[4pt]
\textbf{instance}\ D_1\ \mathit{Int}\ \mathit{Int}\quad \textbf{where}\ \{\mathit{bar} = \mathit{id}\} \\
\textbf{instance}\ D_2\ \mathit{Int}\ \mathit{Bool}\ \textbf{where}\ \{\mathit{baz} = \mathit{even}\}
\end{array}$$

The above instances satisfy the Liberal Coverage Condition and imply that the 3-parameter type class C is inhabited by triples $([\mathit{Int}], [\mathit{Bool}], \mathit{Bool})$ and $([\mathit{Int}], [\mathit{Int}], \mathit{Int})$. If we project the triples on the functional dependency $a \to b$, then we see that $[\mathit{Int}]$ is associated with both $[\mathit{Int}]$ and $[\mathit{Bool}]$. In other words, the functional dependency is violated. Yes, as the following two expressions show, GHC has no qualms about using both instances:

```
ghci> foo [1 :: Int] (True :: Bool)
[False]

ghci> foo [1 :: Int] (2 :: Int)
[1]
```

In short, GHC's current implementation of functional dependencies does not properly enforce the functional dependency property. This is not an implementation problem, but points at problem in the theory: it is an open challenge how to do so under the Liberal Coverage Condition.

8.1.4 Challenge 2: Elaborating Functional Dependencies

GHC elaborates Haskell source programs into the typed intermediate language System $\mathrm{F_C}$ (Sulzmann et al., 2007a), which is an extension of System F with type equality coercions. Unfortunately, when it comes to functional dependencies, the elaboration process is incomplete: while Sulzmann et al. (2007b) provide the most concise and formal account of functional dependencies we are aware of, it has never been investigated how well-typed programs with respect to Sulzmann et al. (2007b) can be elaborated into System $\mathrm{F_C}$.

Hence, GHC currently rejects those programs it cannot elaborate. It turns out, the problem is more general: due to the non-parametric semantics of functional dependencies, it is not possible to translate them to a statically-typed language like System F that features only parametric polymorphism. Indeed, as we discuss in Section 8.6.3, Hugs (which also translates to an intermediate language akin to System F) suffers from the same problem. Consider for instance the following program, which originates from GHC bug report #9627.

$$\begin{array}{ll} \textbf{class}\ C\ a\ b \mid a \to b & f :: C\ \mathit{Int}\ b \Rightarrow b \to \mathit{Bool} \\ \textbf{instance}\ C\ \mathit{Int}\ \mathit{Bool} & f\ x = x \end{array}$$

This program is rejected because GHC has difficulty determining that type b equals *Bool* during the type-checking of function f. Yet, it is actually not difficult to see that the equality holds. From the functional dependency and the one instance for type class C, it follows that *Int* is uniquely associated with *Bool*. Hence, from the type class constraint $C\ \mathit{Int}\ b$, it must indeed follow that b equals *Bool*; b is not a type parameter that can be freely instantiated.

How to elaborate all well-typed Haskell programs with functional dependencies (with respect to the formal system of Sulzmann et al. (2007b)) into a typed intermediate language like System F_C is currently an open problem.

8.1.5 Challenge 3: Deduplicating Functional Dependencies

About ten years ago, a new type-level feature was introduced in Haskell that replicates much of the functionality of functional dependencies: (associated) type families (Chakravarty et al., 2005a). They provide a functional, rather than a relational, notation for expressing a functional dependency between types. For instance, with associated type families we can express the *Coll* type class with a single parameter for the collection type and an associated *Elem* type family for the element type:

$$\begin{array}{l} \textbf{class}\ \mathit{Coll}\ c\ \textbf{where} \\ \quad \textbf{type}\ \mathit{Elem}\ c :: \star \\ \quad \mathit{singleton} :: \mathit{Elem}\ c \to c \\[1ex] \mathit{singleton_2} :: (\mathit{Coll}\ c, \mathit{Coll}\ (\mathit{Elem}\ c)) \Rightarrow \mathit{Elem}\ (\mathit{Elem}\ c) \to c \\ \mathit{singleton_2}\ c = \mathit{singleton}\ (\mathit{singleton}\ c) \end{array}$$

This development means that GHC's Haskell dialect now supports two similar features. This is not necessarily problematic for modeling purposes, because each feature has its notational pros and cons. However, the separate support for both features gives rise to a lot of complexity in the type checker. While there has been a lot of speculation about the comparable expressive power of the two features, no formal comparison has been made. It is still an open engineering challenge to simplify the type checker by sharing the same infrastructure for both features.

8.1.6 Our Approach

The three challenges we have outlined above are all symptoms of a common problem: while we have a formalization of functional dependencies based on *Constraint Handling Rules* (Sulzmann et al., 2007b), we lack a formalization of functional dependencies that captures the functional dependency property properly within the type system and elaborates the feature into System $\mathrm{F_C}$. The former provides a common ground for comparison with associated type families.

This chapter provides such a formalization based on the conjecture of Schrijvers et al. (2007) that functional dependencies can be translated into type families. In terms of the *Coll* example this idea means that we replace the functional dependency annotation $(c \rightarrow e)$ by a new type family *FD* and a superclass constraint $(\mathit{FD}\, c \sim e)$ that captures the functional relation between the c and e parameters.

$$\begin{array}{l}\textbf{class}\ \mathit{FD}\ c \sim e \Rightarrow \mathit{Coll}\ c\ e\ \textbf{where}\\ \quad \mathit{singleton} :: e \rightarrow c\\[1ex] \textbf{type}\ \mathit{FD}\ c :: \star\end{array}$$

Moreover, we derive an appropriate *FD* instance for every *Coll* instance. For example, the list instance:

$$\begin{array}{l}\textbf{instance}\ \mathit{Coll}\ [e]\ e\ \textbf{where}\\ \quad \mathit{singleton}\ x = [x]\end{array}$$

gives rise to the type family instance:

$$\textbf{type}\ \mathit{FD}\ [e] = e$$

Intuitively, this transformation implements an alternative definition of a functional dependency: a relation R satisfies the functional dependency $X \rightarrow Y$, where X and Y are attributes of R, iff

$$\exists f : X \rightarrow Y.\ \forall (x, y) \in R.\ f(x) = y \tag{8.2}$$

This chapter addresses the challenges of functional dependencies with a formalization of the above idea in terms of a fully formal elaboration into System $\mathrm{F_C}$. Our elaboration represents type class dictionaries with GADTs that hold evidence for the functional dependencies. Unlike for other dictionary fields, pattern matching to extract this evidence cannot be encapsulated in projection functions but has to happen at use sites. While GHC already uses this approach in practice for equalities in class contexts, as far as we know this approach has never been formalized before.

Figure 8.1 Extension of Figure 6.1 with Functional Dependencies

$$
\begin{array}{lcll}
F & ::= & \langle \textit{type family name} \rangle & \\[1em]
\textit{cls} & ::= & \textbf{class}\ \forall \overline{a}\overline{b}.\ \overline{\pi} \Rightarrow \texttt{TC}\ \overline{a} \mid \overline{\textit{fd}}^{m}\ \textbf{where}\ \{\ f :: \sigma\ \} & \textit{class decl.} \\
\textit{ins} & ::= & \textbf{instance}\ \forall \overline{a}\overline{b}.\ \overline{\pi} \Rightarrow \texttt{TC}\ \overline{u}\ \textbf{where}\ \{\ f = e\ \} & \textit{instance decl.} \\
\textit{fd} & ::= & a_1 \ldots a_n \rightarrow a_0 & \textit{fundep} \\[1em]
\tau & ::= & a \mid T \mid \tau_1\ \tau_2 \mid F(\overline{\tau}) & \textit{monotype} \\
u & ::= & a \mid T \mid u_1\ u_2 & \textit{type pattern} \\[1em]
\phi & ::= & \tau_1 \sim \tau_2 & \textit{equality constraint} \\
\pi & ::= & \texttt{TC}\ \overline{\tau} & \textit{class constraint} \\
Q & ::= & \phi \mid \pi & \textit{type constraint}
\end{array}
$$

8.2 Logical Reading of Functional Dependencies

Before presenting our formalization of functional dependencies in the next section, this section revisits the logical reading of type classes in the presence of functional dependencies. Class and instance declarations give rise to logical implications (see Section 5.4.2), which we capture in constraint schemes S.

To aid readability, we first present the extensions functional dependencies introduce to the source syntax in Section 8.2.1, and defer the logical interpretation of the class system to Sections 8.2.2 and 8.2.3.

8.2.1 Syntax

The syntax of source programs (*pgm*), declarations (*decl*), expressions (e), qualified types (ρ), polytypes (σ), constraint sets (C), and constraint schemes (S) remains identical to that of the basic system (Figure 6.1). Thus, we only focus on the differences, which are concentrated around monotypes, constraints, and class and instance declarations; Figure 8.1 captures only the syntactic differences between the basic system of Chapter 6 and a system with functional dependencies.

Most interesting programs with functional dependencies require multiple parameters, so the syntax of classes and instances in Figure 8.1 allows for multiple type arguments. Additionally, class declarations can be decorated with functional dependencies *fd*. Functional dependencies take the simple form $a_1 \dots a_n \to a_0$.[3] We also explicitly separate the type variables $\overline{a}$ that appear in the class/instance head from type variables $\overline{b}$ that appear only in the class/instance context $\overline{\pi}$.

We preserve the stratification found in all HM-based systems of types into monotypes, qualified types, and polytypes. Nevertheless, we extend monotypes τ with type family applications $F(\overline{\tau})$ to capture functionally-dependent types. Type families are disallowed in the source text; as we illustrate in the rest of the section, in our formalization each functional dependency gives rise to a type family declaration.

Since monotypes now include type family applications, their syntax no longer coincides with type patterns (as it does in most formalization of type classes found in the literature). Thus, we use different notation for (possibly non-linear) *type patterns* u, which appear in instance declaration heads.

Extensions to the syntax of constraints are also presented in Figure 8.1. Class constraints π take the form $\texttt{TC}\ \overline{\tau}$, to reflect the possibility of multi-parameter type classes. Type constraints Q can be either class constraints or type equality constraints ϕ. Again, type equality constraints cannot appear in the source text; they are introduced by the logical interpretation of functional dependencies.

Finally, as we did for the basic system (Chapter 6), we reduce the notational burden by omitting all mention of kinds and by assuming that each class has exactly one method. Additionally, we assume that all type family applications are fully saturated. In the remainder of the chapter, we denote a type family of arity n as F_n in cases where the arity is of importance.

8.2.2 Logical Reading of Class Declarations

A class declaration of the form

$$\textbf{class}\ \forall \overline{a}\overline{b}.\ \overline{\pi} \Rightarrow \texttt{TC}\ \overline{a} \mid fd_1, \dots, fd_m\ \textbf{where}\ \{\ f :: \sigma\ \}$$

gives rise to two kinds of constraint schemes:

Superclass Constraint Schemes

$$S_{C_\pi} = \forall \overline{a}.\ \texttt{TC}\ \overline{a} \Rightarrow \theta(\pi) \quad \forall \pi \in \overline{\pi} \tag{CS1a}$$

where $\theta = det(\overline{a}, \overline{\pi})$ (we explain the meaning of function det below). This constraint scheme expresses the logical reading of the superclass relation we

[3]We do not consider *multi-range FDs* (Sulzmann et al., 2007b) which can be desugared into simple functional dependencies (Silberschatz et al., 2006).

gave in Section 5.4.2: given a class constraint, we can derive that each of the superclass constraints is also satisfied. As an example, recall classes *Eq* and *Ord* from Section 5.4.1:

$$\begin{array}{ll} \textbf{class } \mathit{Eq}\ a \textbf{ where} & \quad \textbf{class } \mathit{Eq}\ a \Rightarrow \mathit{Ord}\ a \textbf{ where} \\ \quad eq :: a \to a \to \mathit{Bool} & \qquad ge :: a \to a \to \mathit{Bool} \end{array}$$

The *Ord* class gives rise to the superclass constraint scheme:

$$\forall a.\ \mathit{Ord}\ a \Rightarrow \mathit{Eq}\ a$$

As a consequence, we do not have to mention the $\mathit{Eq}\ a$ constraint explicitly in the signature of the function *gt* below. Instead, it can be derived implicitly by the type-checker from the given $\mathit{Ord}\ a$ constraint by means of the scheme.

$$\begin{array}{l} gt :: \mathit{Ord}\ a \Rightarrow a \to a \to \mathit{Bool} \\ gt\ x\ y = ge\ x\ y\ \wedge\ not\ (eq\ x\ y) \end{array}$$

Indeed, this captures the logical interpretation we gave in Section 5.4.2. Yet, functional dependencies complicate matters. Consider deriving the superclass scheme for class *D*:

$$\begin{array}{l} \textbf{class } C\ a\ b \mid a \to b \\ \textbf{class } C\ a\ b \Rightarrow D\ a \end{array}$$

By simply selecting the corresponding type in the class context, we get the following, broken constraint scheme

$$\forall a.\ D\ a \Rightarrow C\ a\ b$$

where b is free! The source of this problem is that b is actually *existentially quantified*; a more appropriate formulation would be:

$$\forall a.\ D\ a \Rightarrow \exists! b.\ C\ a\ b$$

Our language of constraint schemes, which reflects Haskell's type system, does not support top-level existentials though, so such an implication is not directly expressible within the language. Yet, there is a way to express b in terms of the in-scope variable a: given that class C comes with a functional dependency, there *exists* a function symbol (skolem constant) F_C such that $F_C\ a \sim b$ (according to Equation 8.2). Hence, we can substitute b with $F_C\ a$ in the above broken scheme to obtain the valid:

$$\forall a.\ D\ a \Rightarrow C\ a\ (F_C\ a)$$

The computation of such a substitution is performed by function *det*, the formal description of which we defer until Section 8.3.5.

This example makes apparent why such class declarations have been rejected by GHC until now: without a way of explicitly expressing b in terms of a, there is no way to express this relation within the type system.

Functional Dependency Constraint Schemes Every functional dependency $fd_i \equiv a_{i_1} \ldots a_{i_n} \to a_{i_0}$ that accompanies the class corresponds logically to the following constraint scheme:

$$S_{C_{fd_i}} = \forall \overline{a}.\ \mathtt{TC}\ \overline{a} \Rightarrow F_{\mathtt{TC}_i}\ a_{i_1}\ \ldots\ a_{i_n} \sim a_{i_0} \qquad \text{(CS1b)}$$

This constraint scheme directly expresses the functional dependency: given $\mathtt{TC}\ \overline{a}$, we know that *there exists* a function f such that $a_{i_0} = f(a_{i_1}, \ldots, a_{i_n})$. We explicitly give this type-level function for the i-th functional dependency of class $\mathtt{TC}$ the name[4] $F_{\mathtt{TC}_i}$. For example, our running example *Coll* gives rise to one such functional dependency constraint scheme:

$$\forall c.\ \forall e.\ \mathit{Coll}\ c\ e \Rightarrow F_{\mathit{Coll}_1}\ c \sim e$$

Notice how this scheme realizes the first part of the informal transformation of Schrijvers et al. (2007): if we (notionally) replace the functional dependency with a superclass equality constraint, then Scheme CS1b is just a special case of Scheme CS1a.

8.2.3 Logical Reading of Class Instances

A class instance

$$\textbf{instance}\ \forall \overline{a}\overline{b}.\ \overline{\pi} \Rightarrow \mathtt{TC}\ \overline{u}\ \textbf{where}\ \{\ f = e\ \}$$

also yields two kinds of constraint schemes:

Instance Constraint Scheme

$$S_{I_{\overline{\pi}}} = \forall \overline{a}\overline{b}.\ \overline{\pi} \Rightarrow \mathtt{TC}\ \overline{u} \qquad \text{(CS2a)}$$

This constraint scheme directly expresses the logical reading of the class instance: *"If the context $\overline{\pi}$ is satisfied, then ($\mathtt{TC}\,\overline{u}$) also holds".* For example, the list instance of *Eq* yields the scheme:

$$\forall e.\ \mathit{Eq}\ e \Rightarrow \mathit{Eq}\ [e]$$

This interpretation directly reflects the interpretation of plain type classes we gave in Section 5.4.2. Functional dependencies manifest themselves in another constraint scheme, which we discuss next.

[4]As a matter of fact, Jones suggested assigning names to functional dependencies as an interesting extension in his original work (Jones, 2000).

FD Witness Constraint Schemes Every functional dependency $fd_i \equiv a_{i_1} \ldots a_{i_n} \rightarrow s_{i_0}$ that accompanies the class also yields a constraint scheme for the instance:

$$S_{I_{fd_i}} = \forall \overline{c}_i.\ F_{\mathtt{TC}_i}\ u_{i_1}\ \ldots\ u_{i_n} \sim \theta(u_{i_0}) \qquad \text{(CS2b)}$$

where $\overline{c}_i = fv(u_{i_1}, \ldots, u_{i_n})$ and $\theta = det(\overline{c}_i, \overline{\pi})$. The idea of this constraint scheme is to (partly) define the function $F_{\mathtt{TC}_i}$ that witnesses the functional dependency. The partial definition covers the subset of the domain that is covered by the class instance. Other instances give rise to schemes that cover other parts of the function. For example, the following program:

$$\begin{array}{l} \textbf{class}\ \mathtt{TC}\ a\ b \mid a \rightarrow b \\ \textbf{instance}\ \mathtt{TC}\ Int\ Bool \\ \textbf{instance}\ \mathtt{TC}\ (Maybe\ a)\ a \end{array}$$

gives rise to the following FD witness constraint schemes (axioms):

$$\begin{array}{rl} F_{\mathtt{TC}}\ Int & \sim Bool \\ \forall a.\ F_{\mathtt{TC}}\ (Maybe\ a) & \sim a \end{array}$$

Observe that these schemes are essentially type family instances.

Similarly to the superclass constraint schemes, FD witness constraint schemes are quite challenging to derive in the general case. At first sight, it seems easy to determine the right-hand side u_{i_0}: simply take the corresponding parameter in the instance head. This indeed works for the simple examples above, but fails in more advanced cases.

Consider deriving the scheme for the following instance:

$$\textbf{instance}\ \mathtt{TC}\ a\ b \Rightarrow \mathtt{TC}\ [a]\ [b]$$

Simply selecting the corresponding type in the instance head, gives:

$$S_{I''_{fd}} = \forall a.\ F_{\mathtt{TC}}\ [a] \sim [b]$$

This equation is broken again, as type variable b is free. What happens here is that $[b]$ is not determined directly by the other instance argument $[a]$, but indirectly through the instance context ($\mathtt{TC}\ a\ b$). If we apply the FD Constraint scheme to this instance context, we obtain that ($F_{\mathtt{TC}}\ a \sim b$). This equation allows us to express b in terms of a. If we substitute it into the broken equation above, we do obtain a valid defining equation:

$$S_{I''_{fd}} = \forall a.\ F_{\mathtt{TC}}\ [a] \sim [F_{\mathtt{TC}}\ a]$$

Essentially, the FD witness constraint scheme realizes the second part of the transformation of Schrijvers et al. (2007): for every class instance, we generate a new type family instance for each functional dependency of the class.

In general, the derivation of a proper defining equation may require an arbitrary number of such substitution steps. We return to this in Section 8.3.5.

8.3 Type Checking

We now turn to the declarative type system of Haskell with functional dependencies. Our formalization utilizes the syntax of the basic system we gave in Figure 6.1 with the extensions of Figure 8.1.

In contrast to the basic system (Chapter 6) and the other two extensions we have developed (Chapters 7 and 9), for our formalization of functional dependencies we omit the specification of elaboration into System $\mathrm{F_C}$ (for reasons we explain in detail in Section 8.4.4).

To this end, instead of the program theory P, we augment the syntax with the *instance environment* I:

$$I \ ::= \ \bullet \mid I, S \qquad\qquad \textit{instance environment}$$

The instance environment I is pre-populated by the constraint schemes induced by the program's class and instance declarations, and is then extended with *local assumptions* (Vytiniotis et al., 2011) when moving under a qualified type. In essence, the instance environment captures the program theory P without the dictionary variable annotations.

Since most of the system specification remains identical to that of the basic system, the remainder of this section introduces only the new judgments, as well as the changes functional dependencies introduce to the judgments presented in Section 6.3.

8.3.1 Term Typing

Term typing takes the form $I; \Gamma \vdash_{\mathtt{TM}} e : \sigma$ and is given by the same rules as relation $P; \Gamma \vdash_{\mathtt{TM}} e : \sigma \rightsquigarrow t$ of Section 6.3 (modulo their elaboration-related aspects). Additionally, relation $I; \Gamma \vdash_{\mathtt{TM}} e : \sigma$ encompasses Rule TmCast:

$$\frac{I; \Gamma \vdash_{\mathtt{TM}} e : \sigma_1 \qquad I; \Gamma \models \sigma_1 \sim \sigma_2}{I; \Gamma \vdash_{\mathtt{TM}} e : \sigma_2}\ \textsc{TmCast}$$

Essentially, Rule TmCast allows for casting the type of a term e from σ_1 to σ_2, as long as the equality of these types can be established. The satisfiability of the equality constraint is established via the *constraint entailment relation* $I; \Gamma \models S$, which is the focus of Section 8.3.3 below.

8.3.2 Type and Constraint Well-formedness

Type Well-formedness Since for the most part types are the same as for the basic system, we only need to add a new rule to judgment $\Gamma \vdash_{\text{TY}} \sigma \rightsquigarrow \upsilon$, to account for type family applications:

$$\frac{F_n \text{ defined} \qquad \Gamma \vdash_{\text{TY}} \overline{\tau}^n}{\Gamma \vdash_{\text{TY}} F_n(\overline{\tau}^n)} \textsc{WfFam}$$

Given that we omit kind information, Rule WfFam ensures that all type arguments are well-formed and that the family application is fully saturated (by means of index n).

Constraint Well-formedness Well-formedness of constraints takes the form $\Gamma \vdash_{\text{CT}} Q$ and is given by the following rules:

$$\frac{\Gamma \vdash_{\text{TY}} \tau_1 \qquad \Gamma \vdash_{\text{TY}} \tau_2}{\Gamma \vdash_{\text{CT}} \tau_1 \sim \tau_2} \textsc{WfEq} \qquad\qquad \frac{\texttt{TC} \text{ defined} \qquad \Gamma \vdash_{\text{TY}} \overline{\tau}}{\Gamma \vdash_{\text{CT}} \texttt{TC}\ \overline{\tau}} \textsc{WfCls}$$

Rule WfEq ensures that a type equality between types τ_1 and τ_2 is well-formed by checking the well-formedness of types τ_1 and τ_2. Rule WfCls is identical to that of relation $\Gamma \vdash_{\text{CT}} Q \rightsquigarrow \upsilon$ (Section 6.3) of the basic system. The only difference lies in the number of arguments $\overline{\tau}$.

8.3.3 Constraint Entailment

The *constraint entailment relation* takes the form $I; \Gamma \models S$ and is given by the rules presented in Figure 8.2. It can be read as *"given an instance environment I and a typing environment Γ, constraint scheme S holds"*.

Our system needs to check entailment of both type class and equality constraints, which is reflected in its rules: Rules Refl, Trans, Sym and Subst constitute the four standard equality axioms. MP is the modus ponens rule, and Rule Inst instantiates a constraint scheme with a monotype. Rule Spec is the standard axiom rule. Like in Jones' *Constructor Classes* (Jones, 1993), the entailment relation $I; \Gamma \models S$ is transitive, closed under substitution and monotonic (if $I_1; \Gamma \models S$ then $I_1, I_2; \Gamma \models S$).

Figure 8.2 Constraint Entailment Specification

$\boxed{I;\ \Gamma \models S}$ Constraint Entailment

$$\frac{I;\ \Gamma \models \forall a.\ S \qquad \Gamma \vdash_{\mathtt{TY}} \tau}{I;\ \Gamma \models [\tau/a]S}\ \textsc{Inst} \qquad \frac{\Gamma \vdash_{\mathtt{TY}} \tau}{I;\ \Gamma \models \tau \sim \tau}\ \textsc{Refl} \qquad \frac{I;\ \Gamma \models \tau_2 \sim \tau_1}{I;\ \Gamma \models \tau_1 \sim \tau_2}\ \textsc{Sym}$$

$$\frac{I;\ \Gamma \models \tau_1 \sim \tau_2 \qquad I;\ \Gamma \models \tau_2 \sim \tau_3}{I;\ \Gamma \models \tau_1 \sim \tau_3}\ \textsc{Trans}$$

$$\frac{I;\ \Gamma \models Q \Rightarrow S \qquad I;\ \Gamma \models Q}{I;\ \Gamma \models S}\ \mathrm{MP} \qquad \frac{S \in I}{I;\ \Gamma \models S}\ \textsc{Spec}$$

$$\frac{I;\ \Gamma \models [\tau_1/a]Q \qquad I;\ \Gamma \models \tau_1 \sim \tau_2}{I;\ \Gamma \models [\tau_2/a]Q}\ \textsc{Subst}$$

Figure 8.3 Declaration Typing

$\boxed{I;\ \Gamma \vdash_{\mathtt{CLS}} \mathit{cls} : I_c;\ \Gamma_c}$ Class Declaration Typing

$$\frac{\begin{array}{c} \theta = \mathit{det}(\overline{a}, \overline{\pi}) \qquad \mathit{unambig}(\overline{b}, \overline{a}, \overline{\pi}) \qquad \Gamma, \overline{a} \vdash_{\mathtt{TY}} \sigma \\ \mathit{fd}_i \equiv \overline{a}^{i_n} \rightarrow a_{i_0} \qquad I_c = [\overline{\forall \overline{a}.\ \mathtt{TC}\ \overline{a} \Rightarrow \theta(\pi)}, \overline{\forall \overline{a}.\ \mathtt{TC}\ \overline{a} \Rightarrow F_{\mathtt{TC}_i}(\overline{a}^{i_n}) \sim a_{i_0}}^m] \\ \Gamma_c = [f : \forall \overline{a}.\ \mathtt{TC}\ \overline{a} \Rightarrow \sigma] \end{array}}{I;\ \Gamma \vdash_{\mathtt{CLS}} \textbf{class}\ \forall \overline{a}\overline{b}.\ \overline{\pi} \Rightarrow \mathtt{TC}\ \overline{a} \mid \overline{\mathit{fd}}^m\ \textbf{where}\ \{\ f :: \sigma\ \} : I_c;\ \Gamma_c}\ \textsc{Cls}$$

$\boxed{I_c;\ I_i;\ \Gamma \vdash_{\mathtt{INS}} \mathit{ins} : I}$ Instance Declaration Typing

$$\frac{\begin{array}{c} \theta = \mathit{det}(\overline{a}, \overline{\pi}) \qquad \mathit{unambig}(\overline{b}, \overline{a}, \overline{\pi}) \\ (f : \forall \overline{a}.\ \mathtt{TC}\ \overline{a} \Rightarrow \sigma) \in \Gamma \qquad I_c, I_i, \overline{\pi};\ \Gamma, \overline{a}, \overline{b} \vdash_{\mathtt{TM}} e : [\overline{u}/\overline{a}]\sigma \\ \theta_i = \mathit{det}(\mathit{fv}(\overline{u}^{i_n}), \overline{\pi}) \qquad \mathit{fv}(\theta_i(u_{i_0})) \subseteq \mathit{fv}(\overline{u}^{i_n}) \qquad \forall(\mathit{fd}_i \equiv \overline{a}^{i_n} \rightarrow a_{i_0}) \\ I_c, I_i, [\overline{u}/\overline{b}']\overline{\pi};\ \Gamma, \overline{a}, \overline{b} \models [\overline{u}/\overline{b}']Q \qquad \forall(\forall \overline{b}'.\ \mathtt{TC}\ \overline{b}' \Rightarrow Q) \in I_c \\ I = [\overline{\forall \overline{a}_i.\ F_{\mathtt{TC}_i}(\overline{u}^{i_n}) \sim \theta_i(u_{i_0})}^m, \forall \overline{a}.\ \theta(\overline{\pi}) \Rightarrow \mathtt{TC}\ \overline{u}] \end{array}}{I_c;\ I_i;\ \Gamma \vdash_{\mathtt{INS}} \textbf{instance}\ \forall \overline{a}\overline{b}.\ \overline{\pi} \Rightarrow \mathtt{TC}\ \overline{u}\ \textbf{where}\ f = e : I}\ \textsc{Ins}$$

8.3.4 Declaration Typing

Declaration typing appears in Figure 8.3 and conservatively extends that of the basic system (Figure 6.2). Value binding typing ($I;\ \Gamma_1 \vdash_{\mathtt{VAL}} \mathit{val} : \Gamma_2$) and program typing typing ($\vdash_{\texttt{PGM}}$ pgm) are identical to those of the basic system so Figure 8.3 presents only typing for class and instance declarations.

Typing Rules Cls and Ins type check class and instance declarations, respectively, and give rise to the constraint schemes we presented in Sections 8.2.2 and 8.2.3. Both rules differ from the corresponding rules of the basic system in two ways:

1. The *Liberal Coverage Condition* (Sulzmann et al., 2007b, Def. 15) is enforced by the specification ($fv(\theta_i(u_{i_0})) \subseteq fv(\overline{u}^{i_n})$, where $\theta_i = det(fv(\overline{u}^{i_n}), \overline{\pi})$), rather than being an additional, external restriction. This design choice is rather easy to motivate: if the domain of the functional dependency does not determine (even indirectly, via the context) its own image, then the FD has no interpretation as a type-level function.

2. The specification does not accept *ambiguous* class or instance contexts ($unambig(\overline{b}, \overline{a}, \overline{\pi})$). Predicate $unambig$ is defined as:

$$unambig(\overline{b}, \overline{a}, \overline{\pi}) \triangleq \overline{b} \subseteq dom(det(\overline{a}, \overline{\pi}))$$

   For class contexts this restriction ensures the well-formedness of the generated constraint schemes. Similarly, for instance contexts it ensures coherent semantics.

8.3.5 Determinacy Relation

The determinacy relation takes the form $det(\overline{a}, \overline{\pi}) = \theta$ and can be read as *"Given known type variables $\overline{a}$ and a set of local class constraints $\overline{\pi}$, substitution θ maps type variables in $\overline{\pi}$ to equivalent types that draw type variables only from $\overline{a}$".*

Formally, we define $det(\overline{a}, \overline{\pi}) = \theta$ as $\overline{a}; \overline{\pi} \vdash_{\texttt{D}} \bullet \rightsquigarrow^{!} \theta$, where relation $\overline{a}; \overline{\pi} \vdash_{\texttt{D}} \theta_1 \rightsquigarrow \theta_2$ has a single rule:

$$\frac{\begin{array}{c}\texttt{TC}\ \overline{\tau} \in \overline{\pi} \qquad \texttt{TC}\ \overline{a} \mid a_{i_1} \ldots a_{i_n} \to a_{i_0} \\ fv(\tau_{i_0}) \not\subseteq \overline{a} \cup dom(\theta) \qquad fv(\tau_{i_1}, \ldots, \tau_{i_n}) \subseteq \overline{a} \cup dom(\theta)\end{array}}{\overline{a}; \overline{\pi} \vdash_{\texttt{D}} \theta \rightsquigarrow [\overline{Proj_j^T(F_{\texttt{TC}_i}(\theta(\tau_{i_1}), \ldots, \theta(\tau_{i_n})))}/\overline{fv(\tau_{i_0})}] \cdot \theta}\ \textsc{Step}_D$$

We use the exclamation mark (!) to denote repeated applications of Rule $\textsc{Step}_D$, until it does not apply anymore. Note that if the superclass declarations of the program form a *Directed Acyclic Graph* (DAG), then this procedure is terminating.[5]

[5] Readers familiar with the work of Sulzmann et al. (2007b) will recognize that $dom(det(\overline{a}, \overline{\pi})) = closure(\overline{a}, \overline{\pi})$, where $closure(\cdot, \cdot)$ as defined in the *Refined Weak Coverage*

As an example of what the determinacy relation computes, consider the following example from Sulzmann et al. (2007b):

$$\begin{array}{ll} \textbf{class}\ G\ a\ b \mid a \rightarrow b & \textbf{class}\ F\ a\ b \mid a \rightarrow b \\ \textbf{class}\ H\ a\ b \mid a \rightarrow b & \textbf{instance}\ (G\ a\ c, H\ c\ b) \Rightarrow F\ [a]\ [b] \end{array}$$

We compute the set of determined variables $det(a, \{G\ a\ c, H\ c\ b\})$ as follows:

$$\begin{array}{lll} \bullet & \rightsquigarrow\ [F_G(a)/c] & (\text{from } (G\ a\ c)) \\ & \rightsquigarrow\ [F_H(F_G(a))/b, F_G(a)/c] & (\text{from } (H\ c\ b)) \\ & \not\rightsquigarrow & \end{array}$$

To illustrate what the projection type functions $Proj_i^T(\cdot)$ do, let us consider an alternative instance for F:

$$\textbf{instance}\ (G\ a\ (c, Int), H\ c\ b) \Rightarrow F\ [a]\ [b]$$

In this case, we can no longer derive $c \sim F_G(a)$ but rather $(c, Int) \sim F_G(a)$. If we have a type-level function Fst available:

$$\textbf{axiom}\ Fst\ a_1\ a_2 : Fst(a_1, a_2) \sim a_1$$

then c can be expressed in terms of a as: $c \sim Fst(F_G(a))$. In this case, $det(a, \{G\ a\ (c, Int), H\ c\ b\})$ proceeds as follows:

$$\begin{array}{lll} \bullet & \rightsquigarrow\ [Fst(F_G(a))/c] & (\text{from } (G\ a\ (c, Int))) \\ & \rightsquigarrow\ [F_H(Fst(F_G(a)))/b, Fst(F_G(a))/c] & (\text{from } (H\ c\ b)) \\ & \not\rightsquigarrow & \end{array}$$

In general, a projection function $Proj_i^T(\cdot)$ is given by a single axiom

$$\textbf{axiom}\ g\ \overline{a}^n : Proj_i^T(T\ a_1\ \ldots\ a_n) \sim a_i$$

As we illustrate in Appendix B.2, there is no need for such projection axioms, if we equip our system with *kind polymorphism* (Yorgey et al., 2012). Yet, for simplicity, we assume in the remainder of this chapter that such projection functions exist for all data types.

Notice that $det(\overline{a}, \overline{\pi})$ can be non-deterministic (multiple derivations can exist for the same variable). For simplicity, we assume for the remainder of this chapter that it is deterministic, but return to this issue in Section 8.6.

Condition (Sulzmann et al., 2007b, Def. 15). That is, it computes the set of determined variables of $\overline{\pi}$, along with a "proof" of their determinacy.

8.4 Target Language: System F_C

It is clear by now that our logical interpretation of functional dependencies requires type-level functions and explicit handling of type equalities. Thus, plain System F is not sufficient to serve as our target language. Additionally, due to type function clauses being introduced by each class instance, dependently-typed languages like Agda (Norell, 2007) would not be suitable either: we need a target language with support for *open* type functions. Fortunately, such a calculus already exists and constitutes the topic of this section.

8.4.1 Variants of System F_C

System F_C is an extension of System F (Girard, 1972; Reynolds, 1974, 1983) with (a) non-syntactic type equality, and (b) open, non-parametric type functions. System F_C has been originally designed by Sulzmann et al. (2007a) to accommodate a multitude of source-level features including GADTs (Peyton Jones et al., 2006) and type families (Chakravarty et al., 2005a,b; Schrijvers et al., 2008) but has also seen a lot of extensions throughout the years for the accommodation of additional source-level features.[6]

In the remainder of this section, we present a variant of System F_C based on the original (Sulzmann et al., 2007a), which serves as the target of our elaboration algorithm in the next section.

System F_C conservatively extends System F, so the description we provide in this section focuses on the extensions with respect to the contents of Sections 5.2 (plain System F) and 6.1 (System F with datatypes and let-bindings). For a more detailed description of System F_C, we urge the reader to consult the original publication by Sulzmann et al. (2007a).

8.4.2 Syntax and Typing

Types Firstly, System F_C extends System F types to capture type function applications, as well as qualified types. The two new forms are highlighted below:

$$\upsilon ::= a \mid T \mid \upsilon_1\ \upsilon_2 \mid \forall a.\ \upsilon \mid \boxed{F(\overline{\upsilon})} \mid \boxed{\psi \Rightarrow \upsilon} \qquad \textit{type}$$

[6]For example, the work of Weirich et al. (2011) extends System F_C with *roles*, in order to address the unsoundness of the calculus described in #1496, the work of Yorgey et al. (2012) extends System F_C with *kind polymorphism*, and the work of Weirich et al. (2013) extends System F_C with the (in)famous *type-in-type* axiom. More recently, Weirich et al. (2017) introduced System DC, an extension of System F_C with support for dependent types.

Similarly to System F, System F$_C$ is impredicative so the syntax of types does not discriminate between monotypes and type schemes (and—in this case—qualified types). Well-formedness of System F$_C$ types takes the same form as for System F types ($\Gamma \vdash_{\text{TY}} \upsilon$) and the two new forms are handled by the following two rules:

$$\frac{F_n \text{ defined} \qquad \Gamma \vdash_{\text{TY}} \overline{\upsilon}^n}{\Gamma \vdash_{\text{TY}} F_n(\overline{\upsilon}^n)}\ \textsc{TyFam} \qquad\qquad \frac{\Gamma \vdash_{\text{PR}} \psi \qquad \Gamma \vdash_{\text{TY}} \upsilon}{\Gamma \vdash_{\text{TY}} \psi \Rightarrow \upsilon}\ \textsc{TyQual}$$

Rule TyFam handles type family applications, and ensures that (a) the function symbol is in scope, and that (b) the application is fully saturated (for the reason behind this design choice see the work of Sulzmann et al. (2007a, Section 3.6)). Rule TyQual ensures that qualified types of the form ($\psi \Rightarrow \upsilon$) are well-formed, by checking the individual components. Meta-variable ψ represents *proposition types*, which we discuss next.

Propositions Proposition types ψ capture (possibly non-syntactic) equalities between types:

$$\psi \ ::= \ \upsilon_1 \sim \upsilon_2 \qquad\qquad \textit{proposition}$$

Well-formedness for propositions (or type equalities) is straightforward; the judgment takes the form $\Gamma \vdash_{\text{PR}} \psi$ and is given by a single rule:

$$\frac{\Gamma \vdash_{\text{TY}} \upsilon_1 \qquad \Gamma \vdash_{\text{TY}} \upsilon_2}{\Gamma \vdash_{\text{PR}} \upsilon_1 \sim \upsilon_2}\ \textsc{Prop}$$

Just as types classify terms, proposition types are witnessed by *coercions.*

Coercions Denoted by γ, coercions are evidence terms, encoding the proof tree for a type equality. A coercion can take any of the following forms:

$$\begin{array}{rcll} \gamma & ::= & \langle \upsilon \rangle \mid \text{sym}\ \gamma \mid \text{left}\ \gamma \mid \text{right}\ \gamma \mid \gamma_1 \mathbin{;} \gamma_2 \mid \psi \Rightarrow \gamma & \qquad \textit{coercion} \\ & \mid & F(\overline{\gamma}) \mid \forall a.\ \gamma \mid \gamma_1[\gamma_2] \mid g\ \overline{\upsilon} \mid \omega \mid \gamma_1 @ \gamma_2 \mid \gamma_1\ \gamma_2 & \end{array}$$

Reflexivity $\langle \upsilon \rangle$, symmetry (sym γ) and transitivity ($\gamma_1 \mathbin{;} \gamma_2$) express that type equality is an equivalence relation. Syntactic forms $F(\overline{\gamma})$ and ($\gamma_1\ \gamma_2$) capture injection, while (left γ) and (right γ) capture projection, which follows from the injectivity of type application. Equality for universally quantified and qualified types is witnessed by forms $\forall a.\ \gamma$ and $\psi \Rightarrow \gamma$, respectively. Similarly, forms $\gamma_1[\gamma_2]$ and $\gamma_1 @ \gamma_2$ witness the equality of type instantiation or coercion application, respectively.

Additionally, System F_C introduces two new symbol classes: coercion variables and axiom names:

$$\begin{array}{lcl} \omega & ::= & \langle \textit{coercion variable name} \rangle \\ g & ::= & \langle \textit{axiom name} \rangle \end{array}$$

The former represent local constraints and are introduced by explicit coercion abstraction or GADT pattern matching. The latter constitute the axiomatic part of the theory, and are generated from top-level axioms, which correspond to type family instances, newtype declarations (Peyton Jones, 2003), or, as we illustrate in Section 8.5.6, type class instances. Of course, typing environments Γ are extended accordingly:

$$\Gamma \ ::= \ \ldots \mid \Gamma, \omega : \psi \mid \Gamma, g\ \overline{a} : F(\overline{u}) \sim \upsilon \qquad \textit{typing environment}$$

The semantics of coercions we gave above is captured in coercion typing $\Gamma \vdash_{\texttt{CO}} \gamma : \psi$, which is given in Figure 8.4.

Terms Next, System F_C extends System F terms with coercion abstraction $(\Lambda(\omega : \psi).\ t)$, coercion application $(t\ \gamma)$, and explicit type cast $(t \triangleright \gamma)$. The complete syntax of terms with the new forms highlighted is given below:

$$\begin{array}{lcl} t & ::= & x \mid K \mid \Lambda a.\ t \mid t\ \upsilon \mid \lambda(x : \upsilon).\ t \mid t_1\ t_2 \mid \boxed{\Lambda(\omega : \psi).\ t} \qquad \textit{term} \\ & \mid & \boxed{t\ \gamma} \mid \boxed{t \triangleright \gamma} \mid \textbf{case}\ t_1\ \textbf{of}\ \overline{p \to t_2} \mid \textbf{let}\ x : \upsilon = t_1\ \textbf{in}\ t_2 \end{array}$$

Typing for coercion abstraction and application is straightforward and is given by Rules $(\Rightarrow I_\psi)$ and $(\Rightarrow E_\psi)$, respectively:

$$\frac{\omega \notin dom(\Gamma) \qquad \Gamma \vdash_{\texttt{PR}} \psi \qquad \Gamma, \omega : \psi \vdash_{\texttt{TM}} t : \upsilon}{\Gamma \vdash_{\texttt{TM}} \Lambda(\omega : \psi).\ t : \psi \Rightarrow \upsilon}\ (\Rightarrow I_\psi)$$

$$\frac{\Gamma \vdash_{\texttt{TM}} t : \psi \Rightarrow \upsilon \qquad \Gamma \vdash_{\texttt{CO}} \gamma : \psi}{\Gamma \vdash_{\texttt{TM}} t\ \gamma : \upsilon}\ (\Rightarrow E_\psi)$$

More interesting is Rule TmCast, which handles explicit casts of the form $(t \triangleright \gamma)$:

$$\frac{\Gamma \vdash_{\texttt{TM}} t : \upsilon_1 \qquad \Gamma \vdash_{\texttt{CO}} \gamma : \upsilon_1 \sim \upsilon_2}{\Gamma \vdash_{\texttt{TM}} t \triangleright \gamma : \upsilon_2}\ \textsc{TmCast}$$

In simple terms, if a term t has type υ_1 and γ is a witness of the equality $\upsilon_1 \sim \upsilon_2$, then $(t \triangleright \gamma)$ has type υ_2.

Figure 8.4 Coercion Typing

$\boxed{\Gamma \vdash_{\mathsf{co}} \gamma : \psi}$ Coercion Typing

$$\frac{(\omega : \psi) \in \Gamma}{\Gamma \vdash_{\mathsf{co}} c : \psi}\ \textsc{CoVar} \qquad \frac{(g\ \overline{a} : \upsilon_1 \sim \upsilon_2) \in \Gamma \qquad \overline{\Gamma \vdash_{\mathsf{ty}} \upsilon}}{\Gamma \vdash_{\mathsf{co}} g\ \overline{\upsilon} : [\overline{\upsilon}/\overline{a}]\upsilon_1 \sim [\overline{\upsilon}/\overline{a}]\upsilon_2}\ \textsc{CoAx}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \upsilon}{\Gamma \vdash_{\mathsf{co}} \langle \upsilon \rangle : \upsilon \sim \upsilon}\ \textsc{CoRefl} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : \upsilon_1 \sim \upsilon_2}{\Gamma \vdash_{\mathsf{co}} \mathsf{sym}\ \gamma : \upsilon_2 \sim \upsilon_1}\ \textsc{CoSym}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \upsilon_1 \sim \upsilon_2 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \upsilon_2 \sim \upsilon_3\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \mathbin{⨾} \gamma_2 : \upsilon_1 \sim \upsilon_3}\ \textsc{CoTrans} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{ty}} \upsilon_1\ \upsilon_3 \qquad \Gamma \vdash_{\mathsf{co}} \gamma_1 : \upsilon_1 \sim \upsilon_2 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \upsilon_3 \sim \upsilon_4\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1\ \gamma_2 : \upsilon_1\ \upsilon_3 \sim \upsilon_2\ \upsilon_4}\ \textsc{CoApp}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \upsilon_1\ \upsilon_2 \sim \upsilon_3\ \upsilon_4}{\Gamma \vdash_{\mathsf{co}} \mathsf{left}\ \gamma : \upsilon_1 \sim \upsilon_3}\ \textsc{CoL} \qquad \frac{\Gamma \vdash_{\mathsf{co}} \gamma : \upsilon_1\ \upsilon_2 \sim \upsilon_3\ \upsilon_4}{\Gamma \vdash_{\mathsf{co}} \mathsf{right}\ \gamma : \upsilon_2 \sim \upsilon_4}\ \textsc{CoR}$$

$$\frac{F_n \text{ defined} \qquad \overline{\Gamma \vdash_{\mathsf{co}} \gamma : \upsilon_1 \sim \upsilon_2}^{\,n} \qquad \overline{\Gamma \vdash_{\mathsf{ty}} \upsilon_1}^{\,n}}{\Gamma \vdash_{\mathsf{co}} F_n(\overline{\gamma}^n) : F(\overline{\upsilon_1}^n) \sim F(\overline{\upsilon_2}^n)}\ \textsc{CoFam}$$

$$\frac{\begin{array}{c}\Gamma, a \vdash_{\mathsf{co}} \gamma : \upsilon_1 \sim \upsilon_2 \\ \Gamma, a \vdash_{\mathsf{ty}} \upsilon_1 \qquad a \notin \Gamma\end{array}}{\Gamma \vdash_{\mathsf{co}} \forall a.\ \gamma : \forall a.\ \upsilon_1 \sim \forall a.\ \upsilon_2}\ \textsc{CoAll}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \forall a.\ \upsilon_1 \sim \forall a.\ \upsilon_2 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \upsilon_3 \sim \upsilon_4 \qquad \Gamma \vdash_{\mathsf{ty}} \upsilon_3\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1[\gamma_2] : [\upsilon_3/a]\upsilon_1 \sim [\upsilon_4/a]\upsilon_2}\ \textsc{CoIns}$$

$$\frac{\Gamma \vdash_{\mathsf{pr}} \psi \qquad \Gamma \vdash_{\mathsf{co}} \gamma : \upsilon_1 \sim \upsilon_2}{\Gamma \vdash_{\mathsf{co}} \psi \Rightarrow \gamma : (\psi \Rightarrow \upsilon_1) \sim (\psi \Rightarrow \upsilon_2)}\ \textsc{CoQual}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : (\psi \Rightarrow \upsilon_1) \sim (\psi \Rightarrow \upsilon_2) \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \psi}{\Gamma \vdash_{\mathsf{co}} \gamma_1 @ \gamma_2 : \upsilon_1 \sim \upsilon_2}\ \textsc{CoQInst}$$

Patterns Additionally, System F$_C$ patterns include local constraints and existentially-bound type variables, so that they can accommodate GADTs:

$$p \ ::= \ K\ \overline{b}\ (\overline{\omega : \psi})\ (\overline{x : \upsilon}) \qquad \textit{pattern}$$

Variables $\overline{b}$ represent the existentially-quantified type variables of constructor K (variables that do not appear in K's result type). Local constraints $\overline{\psi}$ are annotated with coercion variables $\overline{\omega}$, so that they can be utilized in the right-hand side of a match for explicit type casting. Finally, term variables $\overline{x}$ are explicitly annotated with their types $\overline{\upsilon}$; this allows for linear-time type checking, a highly desirable property for a target language meant to sustain heavy source-to-source transformations.

Though the judgment for pattern typing ($\Gamma \vdash_{\mathrm{P}} p \to t : \upsilon_1 \to \upsilon_2$) is identical to that for System F (Section 6.1), the rule it is given by is extended to take into account the additional constructor fields:

$$\frac{\begin{array}{c}(K : \forall \overline{a}\overline{b}'. \overline{\psi} \Rightarrow \overline{\upsilon} \to T\ \overline{a}) \in \Gamma \qquad \theta = [\overline{\upsilon_a / a}, \overline{b / b'}] \\ \overline{\omega}, \overline{b} \notin \Gamma \qquad \overline{x} \notin dom(\Gamma) \qquad \Gamma, \overline{b}, \overline{(\omega : \theta(\psi))}, \overline{(x : \theta(\upsilon))} \vdash_{\mathrm{TM}} t : \upsilon_2\end{array}}{\Gamma \vdash_{\mathrm{P}} K\ \overline{b}\ \overline{(\omega : \theta(\psi))}\ \overline{(x : \theta(\upsilon))} \to t : T\ \overline{\upsilon}_a \to \upsilon_2}\ \textsc{Pat}$$

Declarations Lastly, System $\mathrm{F_C}$ declarations include data type declarations, type family declarations, top-level equality axioms and value bindings:

$$\begin{array}{rcll} decl & ::= & \textbf{data}\ T\ \overline{a}\ \textbf{where}\ \{\ \overline{K : \upsilon}\ \} & \textit{datatype declaration} \\ & \mid & \textbf{type}\ F(\overline{a}) & \textit{family declaration} \\ & \mid & \textbf{axiom}\ g\ \overline{a} : F(\overline{u}) \sim \upsilon & \textit{equality axiom} \\ & \mid & \textbf{let}\ x : \upsilon = t & \textit{value binding} \end{array}$$

Datatype declarations are similar to those for System F, the only noticeable difference being that data constructors are explicitly annotated with their type (see Section 2.3). Type family declarations are also straightforward and essentially specify the name of the declared open type family, as well as its arity (through type arguments $\overline{a}$).[7] Equality axioms take the form ($\textbf{axiom}\ g\ \overline{a} : F(\overline{u}) \sim \upsilon$) and are a key element of System $\mathrm{F_C}$. Such an axiom axiomatically declares non-syntactic equality between types $F(\overline{u})$ and υ. Type patterns u are defined just as for the source language:

$$u \ ::= \ a \mid T \mid u_1\ u_2 \qquad\qquad \textit{type pattern}$$

Type variables $\overline{a}$ scope over the equality, and axiom name g can be used (appropriately instantiated) to explicitly typecast terms. As an example of type-equality axioms, consider the Peano axioms (Peano, 1889) for addition:

$$\begin{array}{lcl} \textbf{axiom}\ \mathit{add\_zero}\ n & : \mathit{Add}\ (\mathit{Zero}, n) & \sim n \\ \textbf{axiom}\ \mathit{add\_succ}\ n\ m & : \mathit{Add}\ (\mathit{Succ}\ n, m) & \sim \mathit{Succ}\ (\mathit{Add}\ (n, m)) \end{array}$$

[7] If we presented kind information, the declaration would also specify its kind.

Figure 8.5 Declaration Typing

$\boxed{\Gamma_1 \vdash_{\mathtt{D}} decl : \Gamma_2}$ Declaration Typing

$$\frac{\upsilon_{K_i} \equiv \forall \overline{a}\overline{b}_i.\ \overline{\psi}_i \Rightarrow \overline{\upsilon}_i \rightarrow T\ \overline{a} \qquad \Gamma \vdash_{\mathtt{TY}} \upsilon_{K_i}}{\Gamma \vdash_{\mathtt{D}} (\textbf{data}\ T\ \overline{a}\ \textbf{where}\ \{\ \overline{K : \upsilon_K}\ \}) : [T, \overline{K : \upsilon_K}]}\ \textsc{Data}$$

$$\frac{}{\Gamma \vdash_{\mathtt{D}} (\textbf{type}\ F(\overline{a}^n)) : [F_n]}\ \textsc{Family}$$

$$\frac{\Gamma, \overline{a} \vdash_{\mathtt{TY}} u_i \qquad \Gamma, \overline{a} \vdash_{\mathtt{TY}} \upsilon \qquad g \notin dom(\Gamma)}{\Gamma \vdash_{\mathtt{D}} (\textbf{axiom}\ g\ \overline{a} : F(\overline{u}) \sim \upsilon) : [g\ \overline{a} : F(\overline{u}) \sim \upsilon]}\ \textsc{Axiom}$$

$$\frac{\Gamma, x : \upsilon \vdash_{\mathtt{TM}} t : \upsilon \qquad x \notin dom(\Gamma)}{\Gamma \vdash_{\mathtt{D}} (\textbf{let}\ x : \upsilon = t) : [x : \upsilon]}\ \textsc{Value}$$

Axiom *add_zero* states that for any type n type family application $Add\ (Zero, n)$ is axiomatically equal to n. Axiom *add_succ* captures the *Succ*-case.

Value bindings are standard.

Declaration typing takes the form $\Gamma_1 \vdash_{\mathtt{D}} decl : \Gamma_2$ and is given in Figure 8.5. The judgment for program typing takes the same form as for System F extended with datatypes (Section 6.1), hence we omit its definition.

8.4.3 Operational Semantics

The operational semantics for System F$_C$ are given by relation $t_1 \longrightarrow t_2$, which extends the corresponding relation of System F. Since its definition is long and not very relevant to this thesis, we omit it; it can be found in the work of Sulzmann et al. (2007a, Section 3.7).

8.4.4 Meta-theoretical Properties

System F$_C$ exhibits all meta-theoretical properties of System F we presented in Section 5.2.2 (apart from strong normalization), and, notably, type safety and type- and coercion-erasure.

Type Safety Due to System F_C's non-syntactic equality, Sulzmann et al. (2007a) prove soundness under the assumption that the top-level axioms are *consistent.*

Definition 6 (Value Type (Sulzmann et al., 2007a)). *A type υ is a value type if it is of the form $(\forall a.\ \upsilon)$, $(\psi \Rightarrow \upsilon)$, or $(T\ \overline{\upsilon})$.*

Definition 7 (Consistent Typing Environment (Sulzmann et al., 2007a)). *A typing environment Γ is consistent iff:*

1. *If $\Gamma \vdash_{\text{co}} \gamma : (\forall a.\ \upsilon_1) \sim \upsilon_2$ and υ_2 is a value type, then $\upsilon_2 = (\forall a.\ \upsilon_1')$.*
2. *If $\Gamma \vdash_{\text{co}} \gamma : (\psi \Rightarrow \upsilon_1) \sim \upsilon_2$ and υ_2 is a value type, then $\upsilon_2 = (\psi \Rightarrow \upsilon_1')$.*
3. *If $\Gamma \vdash_{\text{co}} \gamma : (T\ \overline{\upsilon}) \sim \upsilon$ and υ is a value type, then $\upsilon = (T\ \overline{\upsilon}')$.*

To our knowledge, the current criterion used by GHC to ensure that typing environments are consistent is captured in the *compatibility relation compat*$(\cdot,\cdot)$, as defined by Eisenberg et al. (2014):

Definition 8 (Compatibility (Eisenberg et al., 2014)). *Two equalities $\phi_1 = F(\overline{u}_1) \sim \tau_1$ and $\phi_2 = F(\overline{u}_1) \sim \tau_2$ are compatible—denoted as compat(ϕ_1, ϕ_2)—iff unify$(\overline{u}_1; \overline{u}_2) = \theta$ implies $\theta(\tau_1) = \theta(\tau_2)$.*

In short, we require that whenever the left-hand sides of two top-level equality axioms overlap, their right-hand sides are equal.

Type and Coercion Erasure The other important property of System F_C is that of type erasure (Theorem 6). That is, the run-time behavior of well-typed System F_C programs is not affected by type or coercion information; both types and coercions can be safely erased.

Though type and coercion erasure is a much desired property, it motivates the lack of the elaboration specification of Section 8.3. The specification of constraint entailment of Figure 8.2 treats equality and class constraints uniformly but their System F_C counterparts belong in different sorts: type equalities ϕ are witnessed by coercions γ, whereas class constraints π are witnessed by System F_C terms t.

What's more, the former are erasable but the latter are not. Thus, a constraint scheme which has a type equality right-hand side has no erasable System F_C counterpart (in contrast to standard constraint schemes which correspond to term-level dictionary transformers).

Though this discrepancy can be addressed, we opted for a simpler and shorter specification; this challenge is addressed in the next section where we discuss our type inference with elaboration algorithm.

8.5 Type Inference and Elaboration into System F$_C$

This section explains how to infer (principal) types for source language programs with functional dependencies and how to elaborate them into System F$_C$ at the same time. Before presenting the details of the type inference algorithm, Section 8.5.1 first presents some additional constructs that are used by the algorithm.

8.5.1 Additional Constructs

During elaboration, we use the following additional constructs.

Type and Evidence Substitutions First, we extend evidence substitutions η (Section 6.4) to accommodate evidence terms for type equality constraints:

$$\eta ::= \bullet \mid [t/d] \cdot \eta \mid [\gamma/\omega] \cdot \eta \qquad \textit{evidence substitution}$$

Type substitutions are standard, but to avoid extending all relations with untouchable variables $\overline{a}$ (see Section 6.4), for this chapter we explicitly denote unification variables by Greek letters α and β, and skolem variables by English letters a and b. Thus, type substitutions take the following form:

$$\theta ::= \bullet \mid [\tau/\alpha] \cdot \theta \qquad \textit{type substitution}$$

Evidence Annotations Additionally, we lift the instance environment I to the program theory P, by annotating all evidence terms (equalities ϕ and class constraints π) with their corresponding System F$_C$ evidence variable:

$$P ::= \bullet \mid P, g\ \overline{a} : F(\overline{u}) \sim \tau \mid P, \omega : \phi \mid P, d : \pi \mid P, d : \forall \overline{a}.\ \overline{\pi} \Rightarrow \texttt{TC}\ \overline{u}$$

Notice that dictionary variables d are not associated with constraint schemes S; the definition of constraint schemes allows a type equality to appear in the right-hand side of an implication. Due to the reasons we discussed in Section 8.4.4, our algorithm treats type equalities differently.

Similarly, we annotate constraints with System F$_C$ variables of the corresponding sort:

$$\begin{array}{lcll}
\mathcal{E} & ::= & \bullet \mid \mathcal{E}, \omega : \phi & \textit{annotated type equalities} \\
\Pi & ::= & \bullet \mid \Pi, d : \pi & \textit{annotated class constraints} \\
\mathcal{Q} & ::= & \omega : \phi \mid d : \pi & \textit{annotated type constraint} \\
\mathcal{C} & ::= & \bullet \mid \mathcal{C}, \mathcal{Q} & \textit{annotated type constraints}
\end{array}$$

Match Contexts We also introduce *match contexts* $\mathbb{E}$, that is, nested case expressions with a hole.

$$\mathbb{E} ::= \Box \mid \textbf{case } d \textbf{ of } p \to \mathbb{E} \qquad \textit{match context}$$

Match contexts are introduced via *dictionary destruction*, denoted as $\Pi \Downarrow \mathbb{E}$ which we define as follows:

$$\frac{}{\bullet \Downarrow \Box}\ \textsc{Empty}$$

$$\frac{\begin{array}{c} K_{\texttt{TC}} : \forall \overline{a}\overline{b}.\ \overline{\psi} \Rightarrow \overline{\tau} \to \upsilon \to T\ \overline{a} \\ \overline{b}', \overline{\omega}, \overline{d}, f \text{ fresh} \qquad \theta = [\overline{\upsilon}'/\overline{a}, \overline{b}'/\overline{b}] \qquad \overline{d : \theta(\tau)}, \Pi \Downarrow \mathbb{E}_2 \\ \mathbb{E} = \textbf{case } d_a \textbf{ of } K_{\texttt{TC}}\ \overline{b}'\ \overline{(\omega : \theta(\psi))}\ \overline{(d : \theta(\tau))}\ (f : \theta(\upsilon)) \to \mathbb{E}_2 \end{array}}{(d_a : T_{\texttt{TC}}\ \overline{\upsilon}'), \Pi \Downarrow \mathbb{E}}\ (\Downarrow)$$

Dictionary destruction $\Pi \Downarrow \mathbb{E}$ recursively pattern matches against class dictionaries Π in a depth-first fashion, thus exposing all superclass constraints and FD-induced type equalities. In short, it computes the transitive closure of the superclass relation.

Throughout the remainder of the chapter, we denote the evidence or typing bindings introduced by a match context $\mathbb{E}$ as $P_{\mathbb{E}}$ or $\Gamma_{\mathbb{E}}$, respectively. Function $\textsc{binds}(\mathbb{E}) = \Gamma; P$ below illustrates how the bindings can be extracted from a match context (though we do not show it in the remainder of the chapter, to avoid additional clutter):

$$\begin{array}{ll} \textsc{binds}(\Box) & = \bullet; \bullet \\ \textsc{binds}(\textbf{case } d \textbf{ of } p \to \mathbb{E}) & = (\overline{b}, \overline{(f : \upsilon)}, \Gamma); (\overline{(\omega : \psi)}, \overline{(d : \tau)}, P) \\ \quad \textbf{where} \quad p & \equiv K_{\texttt{TC}}\ \overline{b}\ \overline{(\omega : \psi)}\ \overline{(d : \tau)}\ \overline{(f : \upsilon)} \\ \qquad\qquad \Gamma; P = \textsc{binds}(\mathbb{E}) & \end{array}$$

8.5.2 Term Elaboration

Figure 8.6 presents type inference and elaboration of terms into System F_C. The judgment takes the form $\Gamma \vdash_{\text{TM}} e : \tau \rightsquigarrow t \mid \Pi; \mathcal{E}$. Given a typing environment Γ and a term e, it computes a set of *wanted* class constraints Π, a set of pending equality constraints $\mathcal{E}$, a monotype τ, and a System F_C term t.

Compare this judgment to that of the basic system: $\Gamma \vdash_{\text{TM}} e : \tau \rightsquigarrow t \mid \mathcal{C}\ ; E$ (Figure 6.3). Since System F_C allows for explicit type equality, wanted equality constraints E are now lifted to coercion-variable-annotated equalities $\mathcal{E}$. Similarly, wanted class constraints take the more restrictive form Π; all wanted equality constraints are captured in $\mathcal{E}$.

Figure 8.6 Term Elaboration

$\boxed{\Gamma \vdash_{\texttt{TM}} e : \tau \rightsquigarrow t \mid \Pi; \mathcal{E}}$ Term Elaboration

$$\frac{(x : \forall \overline{a}\,\overline{b}.\ \overline{\pi} \Rightarrow \tau) \in \Gamma \qquad \overline{\alpha}, \overline{d}\ \text{fresh} \qquad \theta = [\overline{\alpha}/\overline{a}] \cdot det(\overline{\alpha}, \overline{\pi})}{\Gamma \vdash_{\texttt{TM}} x : \theta(\tau) \rightsquigarrow x\ \overline{\alpha}\ \overline{(\theta(\overline{b}))}\ \overline{d} \mid \overline{(d : \theta(\pi))}; \bullet}\ \text{TmVar}$$

$$\frac{\Gamma, x : \alpha \vdash_{\texttt{TM}} e : \tau \rightsquigarrow t \mid \Pi; \mathcal{E} \qquad \alpha\ \text{fresh}}{\Gamma \vdash_{\texttt{TM}} \lambda x.\ e : (\alpha \to \tau) \rightsquigarrow \lambda(x : \alpha).\ t \mid \Pi; \mathcal{E}}\ \text{TmAbs}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\texttt{TM}} e_1 : \tau_1 \rightsquigarrow t_1 \mid \Pi_1; \mathcal{E}_1 \qquad \Gamma \vdash_{\texttt{TM}} e_2 : \tau_2 \rightsquigarrow t_2 \mid \Pi_2; \mathcal{E}_2 \\ \alpha, \omega\ \text{fresh} \qquad \Pi = \Pi_1, \Pi_2 \qquad \mathcal{E} = \mathcal{E}_1, \mathcal{E}_2, \omega : \tau_1 \sim \tau_2 \to \alpha\end{array}}{\Gamma \vdash_{\texttt{TM}} e_1\ e_2 : a \rightsquigarrow (t_1 \rhd \omega)\ t_2 \mid \Pi; \mathcal{E}}\ \text{TmApp}$$

$$\frac{\begin{array}{c}\Gamma, x : \alpha \vdash_{\texttt{TM}} e_1 : \tau_1 \rightsquigarrow t_1 \mid \Pi_1; \mathcal{E}_1 \qquad \Gamma, x : \tau_1 \vdash_{\texttt{TM}} e_2 : \tau_2 \rightsquigarrow t_2 \mid \Pi_2; \mathcal{E}_2 \\ \alpha, \omega\ \text{fresh} \qquad \Pi = \Pi_1, \Pi_2 \qquad \mathcal{E} = \mathcal{E}_1, \mathcal{E}_2, \omega : \alpha \sim \tau_1\end{array}}{\Gamma \vdash_{\texttt{TM}} (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : \tau_2 \rightsquigarrow (\textbf{let}\ x : elab_{\texttt{TY}}(\tau_1) = t_1\ \textbf{in}\ t_2) \mid \Pi; \mathcal{E}}\ \text{TmLet}$$

The differences between the rules of Figure 8.6 with the corresponding rules of the basic system (Figure 6.3) are highlighted.

The most interesting rule is TmVar, which handles variables. We denote by $\overline{a}$ the type variables that appear in τ, and by $\overline{b}$ the ones that appear only in the context $\overline{\pi}$. The rule introduces *wanted* class constraints, appropriately instantiated with fresh unification and dictionary variables.

Notice that, unlike $\overline{a}$, variables $\overline{b}$ are not instantiated with fresh unification variables. Instead, we use the determinacy relation to express them in terms of $\overline{a}$. For example, given **class** $D\ a\ b \mid a \to b$, and $(x : D\ a\ b \Rightarrow a \to a) \in \Gamma$, we infer for x type $\alpha \to \alpha$, giving rise to the wanted constraint $D\ \alpha\ (F_D\ \alpha)$.

This treatment of $\overline{b}$ allows the unification algorithm of the next section to indirectly refine the non-parametric parameters of class constraints. Of course, this requires that x's signature is *unambiguous*, an issue we return to in Section 8.6.

Rule TmAbs is entirely standard.

Rule TmApp handles term applications $(e_1\ e_2)$. In addition to the constraints introduced by each subterm, we also require that $(\tau_1 \sim \tau_2 \to \alpha)$, like all HM(X)-based systems. In order to ensure that the elaborated term is well-typed, we explicitly cast e_1 with ω, which serves as a placeholder for the equality proof computed by the *constraint entailment relation* (Section 8.5.5).

Rule TmLet handles (possibly recursive) let bindings. The only difference between the rule and the corresponding one for the basic system is concentrated in coercion variable ω, which captures the requirement that x has a single type (but is otherwise ignored).

8.5.3 Type, Constraint, and Environment Elaboration

Constraint Elaboration Since equalities and class constraints correspond to different System $\mathrm{F_C}$ syntactic sorts, constraint elaboration is performed by two separate relations. First, class constraints π are elaborated into dictionary types, via function $elab_{\mathtt{CC}}(\pi) = \upsilon$, given by the following clause:

$$elab_{\mathtt{CC}}(\mathtt{TC}\ \overline{\tau}) = T_{\mathtt{TC}}\ \overline{elab_{\mathtt{TY}}(\tau)}$$

Second, type equalities ϕ are elaborated into System $\mathrm{F_C}$ proposition types ψ, via function $elab_{\mathtt{EQ}}(\phi) = \psi$, given by the following clause:

$$elab_{\mathtt{EQ}}(\tau_1 \sim \tau_2) = elab_{\mathtt{TY}}(\tau_1) \sim elab_{\mathtt{TY}}(\tau_2)$$

Type Elaboration Elaboration of types is given by function $elab_{\mathtt{TY}}(\sigma) = \upsilon$. Since System $\mathrm{F_C}$ types conservatively extend System F types, we only present the clauses which handle the new syntactic forms:

$$\begin{array}{lcl} elab_{\mathtt{TY}}(F_n(\overline{\tau}^n)) & = & F_n(\overline{elab_{\mathtt{TY}}(\tau)}^n) \\ elab_{\mathtt{TY}}(\overline{\phi} \Rightarrow \overline{\pi} \Rightarrow \tau) & = & \overline{elab_{\mathtt{EQ}}(\phi)} \Rightarrow \overline{elab_{\mathtt{CC}}(\pi)} \rightarrow elab_{\mathtt{TY}}(\tau) \end{array}$$

First, the elaboration of a type family application is straightforward. Second, since the order of constraints in a qualified type given in prenex normal form is irrelevant, the last clause of $elab_{\mathtt{TY}}(\cdot)$ handles qualified types in an uncurried way. Hence, a type in prenex normal form in the source language gets translated into a System $\mathrm{F_C}$ type in prenex normal form (otherwise, the elaboration can result in higher-rank types where qualification can appear in nested positions).

Environment Elaboration Finally, in order to state the correctness of our algorithm in Section 8.6, we define environment elaboration functions $elab_{\mathtt{PT}}(P) = \Gamma'$ and $elab_{\mathtt{TE}}(\Gamma) = \Gamma'$, which translate source-level environments (program theory P and typing environment Γ, respectively) into System $\mathrm{F_C}$ typing environments

Γ'. Both definitions are straightforward. The former is given by the following clauses:

$$\begin{array}{lcl}
elab_{\text{PT}}(\bullet) & = & \bullet \\
elab_{\text{PT}}(P, g\ \overline{a} : F(\overline{u}) \sim \tau) & = & elab_{\text{PT}}(P), g\ \overline{a} : F(\overline{u}) \sim elab_{\text{TY}}(\tau) \\
elab_{\text{PT}}(P, \omega : \phi) & = & elab_{\text{PT}}(P), \omega : elab_{\text{EQ}}(\phi) \\
elab_{\text{PT}}(P, d : \pi) & = & elab_{\text{PT}}(P), d : elab_{\text{CC}}(\pi) \\
elab_{\text{PT}}(P, d : \forall \overline{a}.\ \overline{\pi} \Rightarrow \texttt{TC}\ \overline{u}) & = & elab_{\text{PT}}(P), d : \forall \overline{a}.\ \overline{elab_{\text{CC}}(\pi)} \rightarrow T_{\texttt{TC}}\ \overline{u}
\end{array}$$

The latter is identical to that of the basic system and is hence omitted.

8.5.4 Type Unification

We now turn to type unification in the presence of functional dependencies.

Type Reduction Judgment $P \models_{\text{R}} \tau \rightsquigarrow \tau'; \gamma$ defines a (single-step) *type reduction relation* on monotypes, specified by the following rules.

$$\frac{P \models_{\text{R}} \tau_1 \rightsquigarrow \tau_1'; \gamma}{P \models_{\text{R}} \tau_1\ \tau_2 \rightsquigarrow \tau_1'\ \tau_2; \gamma\ \langle \tau_2 \rangle}\ \textsc{Left}_R \qquad \frac{P \models_{\text{R}} \tau_2 \rightsquigarrow \tau_2'; \gamma}{P \models_{\text{R}} \tau_1\ \tau_2 \rightsquigarrow \tau_1\ \tau_2'; \langle \tau_1 \rangle\ \gamma}\ \textsc{Right}_R$$

$$\frac{P \models_{\text{R}} \tau_i \rightsquigarrow \tau_i'; \gamma_i \qquad \tau_j' = \tau_j, \forall j \neq i}{P \models_{\text{R}} F(\overline{\tau}^n) \rightsquigarrow F(\overline{\tau}'^n); F(\langle \tau_1 \rangle, \cdots \gamma_i, \cdots \langle \tau_n \rangle)}\ \textsc{Arg}_R$$

$$\frac{(g\ \overline{a} : F(\overline{u}) \sim \tau) \in P}{P \models_{\text{R}} [\overline{\tau}/\overline{a}]F(\overline{u}) \rightsquigarrow [\overline{\tau}/\overline{a}]\tau; g\ \overline{\tau}}\ \textsc{Axiom}_R$$

We perform type reduction under program theory P, such that Rule $\textsc{Axiom}_R$ can expand type family applications when an appropriate axiom matches. We also annotate the reduction with a coercion γ, which witnesses the equality $\tau \sim \tau'$, as is required by the unification relation which we discuss next. Type reduction is sound, which can be proven by straightforward induction on the type reduction derivation:

Lemma 1 (Soundness of Type Reduction)**.** *If* $\Gamma \models_{\text{TY}} \tau_1$ *and* $P \models_{\text{R}} \tau_1 \rightsquigarrow \tau_2; \gamma$, *then* $elab_{\text{TE}}(\Gamma), elab_{\text{PT}}(P) \vdash_{\text{CO}} \gamma : elab_{\text{TY}}(\tau_1) \sim elab_{\text{TY}}(\tau_2)$.

Type Unification Type reduction is used by the *single-step unification relation* $P \vdash_{\mathtt{U}} \omega : \tau_1 \sim \tau_2 \rightsquigarrow \mathcal{E};\ \theta;\ \eta$, which is given by the following rules:

$$\frac{}{P \vdash_{\mathtt{U}} \omega : \tau \sim \tau \rightsquigarrow \bullet;\ \bullet;\ [\langle\tau\rangle/\omega]}\ \textsc{Refl}_U$$

$$\frac{P \vdash_{\mathtt{U}} \omega' : \tau_2 \sim \tau_1 \rightsquigarrow \mathcal{E};\ \theta;\ \eta \qquad \omega' \text{ fresh}}{P \vdash_{\mathtt{U}} \omega : \tau_1 \sim \tau_2 \rightsquigarrow \mathcal{E};\ \theta;\ \eta \cdot [\text{sym } \omega'/\omega]}\ \textsc{Sym}_U$$

$$\frac{\alpha \notin fv(\tau)}{P \vdash_{\mathtt{U}} \omega : \alpha \sim \tau \rightsquigarrow \bullet;\ [\tau/\alpha];\ [\langle\tau\rangle/\omega]}\ \textsc{Var}_U$$

$$\frac{P \vdash_{\mathtt{R}} F(\overline{\tau}) \rightsquigarrow \tau_2;\ \gamma \qquad \omega' \text{ fresh}}{P \vdash_{\mathtt{U}} \omega : F(\overline{\tau}) \sim \tau_1 \rightsquigarrow \{\omega' : \tau_2 \sim \tau_1\};\ \bullet;\ [\gamma \mathbin{;} \omega'/\omega]}\ \textsc{Red}_U$$

$$\frac{\omega_1, \omega_2 \text{ fresh} \qquad \gamma = \omega_1\ \omega_2}{P \vdash_{\mathtt{U}} \omega : \tau_1\ \tau_2 \sim \tau_1'\ \tau_2' \rightsquigarrow \{\omega_1 : \tau_1 \sim \tau_1', \omega_2 : \tau_2 \sim \tau_2'\};\ \bullet;\ [\gamma/\omega]}\ \textsc{App}_U$$

In short, the judgment holds for an equality $\tau_1 \sim \tau_2$ iff the unification problem can be reduced to a simpler unification problem for the set of equality constraints $\mathcal{E}$ and type substitution θ. Since in our target language casting needs explicit equality proofs, we also accumulate an evidence substitution η, which explains how evidence for $\mathcal{E}$ can be turned into evidence for $\tau_1 \sim \tau_2$.

8.5.5 Constraint Entailment

Single-step constraint entailment takes the form $P \vdash_{\mathtt{E}} \mathcal{Q} \rightsquigarrow \mathcal{C};\ \theta;\ \eta$ and simplifies a constraint $\mathcal{Q}$ to a set of simpler constraints $\mathcal{C}$ and a type substitution θ. Additionally, it computes an evidence substitution η, which maps evidence variables (coercion or dictionary variables) to evidence terms composed by the simpler evidence. The relation is given in Figure 8.7.

Our system needs to handle both class and equality constraints, which is reflected in the rules: Rule $\textsc{Cls}_E$ formalizes the standard SLD resolution (backwards chaining), Rule $\textsc{Eq}_E$ performs single-step unification on equality constraints, and Rule $\textsc{Red}_E$ allows for type reduction on class parameters.

Figure 8.7 Constraint Entailment

$\boxed{P \vdash_{\mathtt{E}} Q \rightsquigarrow C;\ \theta;\ \eta}$ Constraint Entailment (Single-step)

$$\frac{(d' : \forall \overline{a}.\ \overline{\pi} \Rightarrow \mathtt{TC}\ \overline{u}) \in P \qquad \overline{d} \text{ fresh}}{P \vdash_{\mathtt{E}} d : [\overline{\tau}/\overline{a}](\mathtt{TC}\ \overline{u}) \rightsquigarrow (\overline{d : [\overline{\tau}/\overline{a}]\pi});\ \bullet;\ [d'\ \overline{\tau}\ \overline{d}/d]}\ \textsc{Cls}_E$$

$$\frac{P \vdash_{\mathtt{U}} \omega : \tau_1 \sim \tau_2 \rightsquigarrow \mathcal{E};\ \theta;\ \eta}{P \vdash_{\mathtt{E}} \omega : \tau_1 \sim \tau_2 \rightsquigarrow \mathcal{E};\ \theta;\ \eta}\ \textsc{Eq}_E$$

$$\frac{P \vdash_{\mathtt{R}} \tau_i \rightsquigarrow \tau_i';\ \gamma_i \qquad \forall i \in [1 \ldots n] \qquad d' \text{ fresh}}{P \vdash_{\mathtt{E}} (d : \mathtt{TC}\ \overline{\tau}^n) \rightsquigarrow (d' : \mathtt{TC}\ \overline{\tau'}^n);\ \bullet;\ [d' \triangleright T_{\mathtt{TC}}\ \overline{\mathsf{sym}\ \gamma_i}^n/d]}\ \textsc{Red}_E$$

By repeatedly applying single-step constraint entailment, we obtain the reflexive and transitive closure $P \vdash_{\mathtt{E}} \mathcal{C} \rightsquigarrow^* \mathcal{C};\ \theta;\ \eta$:

$$\frac{}{P \vdash_{\mathtt{E}} \mathcal{C} \rightsquigarrow^* \mathcal{C};\ \bullet;\ \bullet}\ \textsc{Stop}_E$$

$$\frac{P \vdash_{\mathtt{E}} \mathcal{Q} \rightsquigarrow \mathcal{C}_1;\ \theta_1;\ \eta_1 \qquad P \vdash_{\mathtt{E}} \theta_1(\mathcal{C}), \mathcal{C}_1 \rightsquigarrow^* \mathcal{C}_2;\ \theta_2;\ \eta_2}{P \vdash_{\mathtt{E}} \mathcal{C}, \mathcal{Q} \rightsquigarrow^* \mathcal{C}_2;\ (\theta_2 \cdot \theta_1);\ (\eta_2 \cdot \eta_1)}\ \textsc{Step}_E$$

We denote the case when $\mathcal{C}$ cannot be further reduced as $P \vdash_{\mathtt{E}} \mathcal{C} \rightsquigarrow^! \mathcal{C}';\ \theta;\ \eta$. To ensure that type inference is decidable, it is essential that constraint entailment is terminating; Section 8.6 provides sufficient conditions.

8.5.6 Declaration Elaboration

Finally, we now turn to type inference and elaboration of top-level declarations. Only elaboration of class and instance declarations is given. Value typing is given by judgment $P;\ \Gamma \vdash_{\mathtt{VAL}} \mathit{val} : \Gamma_v \rightsquigarrow \mathit{decl}$ and its definition is straightforward; it is given in Appendix B.3.

Elaboration of Class Declarations

Class elaboration takes the form $\Gamma \vdash_{\mathtt{CLS}} \mathit{cls} \rightsquigarrow \overline{\mathit{decl}} \mid \Gamma_c$ and is given by a single rule, Rule Cls (Figure 8.8).

The encoding of a class constraint in System F_C is that of a GADT-dictionary (Peyton Jones et al., 2006), such that we can store existentially

Figure 8.8 Class Declaration Elaboration

$\boxed{\Gamma \vdash_{\texttt{CLS}} cls \rightsquigarrow \overline{decl} \mid \Gamma_c}$ Class Elaboration

$$\dfrac{\begin{array}{ccc} unambig(\overline{b}, \overline{a}, \overline{\pi}) & \Gamma, \overline{a} \vdash_{\texttt{TY}} \sigma \rightsquigarrow \upsilon & \overline{\Gamma, \overline{a}, \overline{b} \vdash_{\texttt{CC}} \pi \rightsquigarrow \tau} \\ \psi_i = F_{\texttt{TC}_i}(\overline{a}^{i_n}) \sim a_{i_0} & fd_i \equiv \overline{a}^{i_n} \to a_{i_0} & \Gamma_c = [f : \forall \overline{a}.\ \texttt{TC}\ \overline{a} \Rightarrow \sigma] \\ \multicolumn{3}{l}{\begin{array}{rcl} decl_1 & = & \mathbf{data}\ T_{\texttt{TC}}\ \overline{a}\ \mathbf{where}\ \{\ K_{\texttt{TC}} : \forall \overline{a}\overline{b}.\ \overline{\psi} \Rightarrow \overline{\tau} \to \upsilon \to T_{\texttt{TC}}\ \overline{a}\ \} \\ \overline{decl_2} & = & \overline{\mathbf{type}\ F_{\texttt{TC}_i}\ \overline{a}^{i_n}}^{m} \\ decl_3 & = & f = \Lambda \overline{a}.\ \lambda(d : T_{\texttt{TC}}\ \overline{a}).\ \mathbf{case}\ d\ \mathbf{of}\ \{\ K_{\texttt{TC}}\ \overline{b}\ \overline{\omega}\ \overline{d}\ x \to x\ \} \end{array}} \end{array}}{\Gamma \vdash_{\texttt{CLS}} \mathbf{class}\ \forall \overline{a}\overline{b}.\ \overline{\pi} \Rightarrow \texttt{TC}\ \overline{a} \mid \overline{fd}^{m}\ \mathbf{where}\ f :: \sigma \rightsquigarrow [decl_1, \overline{decl_2}, decl_3] \mid \Gamma_c}\ \textsc{Cls}$$

quantified variables $\overline{b}$, as well as the local constraints ψ_i, each corresponding to a functional dependency annotation.

In contrast to earlier formalizations of type classes, checking a class declaration does not give rise to a direct extension of the program theory. While Equation CS1a may have a direct interpretation as a System F_C term, Equation CS1b does not: System F_C does not support functions that return coercions; this would not be compatible with System F_C's coercion erasure and a call-by-need semantics.

Instead, both schemes can be uniformly elaborated as *match contexts*. For example, the following match context corresponds to the logical implication $Ord\ a \Rightarrow Eq\ a$:

$$\mathbb{E} = \mathbf{case}\ d_{Ord}\ \mathbf{of}\ \ \{\ K_{Ord}\ d_{Eq}\ f \to \Box\ \}$$

We reject unconditionally ambiguous class declarations, via restriction $unambig(\overline{b}, \overline{a}, \overline{\pi})$.

Elaboration of Class Instances

Instance elaboration takes the form $P; \Gamma \vdash_{\texttt{INS}} ins \rightsquigarrow \overline{decl} \mid P_i$ and appears in Figure 8.9. An instance declaration *ins* is elaborated to System F_C declarations $\overline{decl}$ and gives rise to the program theory extension P_i. To aid readability, we formalize instance elaboration by means of the following auxiliary relations:

Axiom Generation As we explained in Section 8.2.3, each class instance gives rise to a type family axiom for every functional dependency of the class. This semantics is reflected in Equation CS2b, and directly corresponds to axioms

Figure 8.9 Class Instance Elaboration

$\boxed{P; \Gamma \vdash_{\mathtt{INS}} ins \rightsquigarrow \overline{decl} \mid P_i}$ Instance Elaboration

$$\dfrac{\begin{array}{c} P_I = P, P_{ax}, \overline{d : \pi}, P_{\mathbb{E}} \quad \begin{array}{c} unambig(\bar{b}, \bar{a}, \overline{\pi}) \quad (\overline{d : \pi}) \Downarrow \mathbb{E} \\ \Gamma_I = \Gamma, \bar{a}, \bar{b}, \Gamma_{\mathbb{E}} \\ \hline \overline{\Gamma, \bar{a}, \bar{b} \vdash_{\mathtt{CC}} \pi \rightsquigarrow \tau} \end{array} \quad \begin{array}{c} d_I, \bar{d} \text{ fresh} \\ S_I = \forall \bar{a}\bar{b}.\, \overline{\pi} \Rightarrow \mathtt{TC}\ \overline{u} \end{array} \\ (f : \forall \overline{a}'.\, \mathtt{TC}\ \overline{a}' \Rightarrow \sigma) \in \Gamma \qquad P_I, d_I : S_I; \Gamma_I \vdash_{\mathtt{TM}} e : [\overline{u}/\overline{a}']\sigma \rightsquigarrow t \\ S_I \hookrightarrow P_{ax} \qquad P_I \vdash_{\mathtt{SC}} \mathtt{TC}\ \overline{u} \rightsquigarrow (\overline{\tau}_b, \bar{t}_d, \overline{\gamma}_c) \qquad P_i = P_{ax}, d_I : \forall \bar{a}\bar{b}.\, \overline{\pi} \Rightarrow \mathtt{TC}\ \overline{u} \\ \begin{array}{rcl} \overline{decl}_1 & = & \textbf{axiom}\ P_{ax} \\ decl_2 & = & \textbf{let}\ d_I = \Lambda \bar{a}\bar{b}.\, \lambda \overline{(d : \tau)}.\, \mathbb{E}[K_{\mathtt{TC}}\ \overline{\tau}_b\ \bar{t}_d\ \overline{\gamma}_c\ t] \end{array} \end{array}}{P; \Gamma \vdash_{\mathtt{INS}} \textbf{instance}\ \forall \bar{a}\bar{b}.\, \overline{\pi} \Rightarrow \mathtt{TC}\ \overline{u}\ \textbf{where}\ f = e \rightsquigarrow [\overline{decl}_1, decl_2] \mid P_i} \ \textsc{Ins}$$

P_{ax}, as produced by relation:

$$\dfrac{\overline{\begin{array}{c} (fd_i \equiv \overline{a}^{i_n} \to a_{i_0}) \in (\overline{fd}^m \in \mathtt{TC}) \\ \theta_i = det(fv(\overline{u}^{i_n}), \overline{\pi}) \qquad fv(\theta_i(u_{i_0})) \subseteq fv(\overline{u}^{i_n}) \qquad \overline{g} \text{ fresh} \end{array}}}{(\forall \bar{a}\bar{b}.\, \overline{\pi} \Rightarrow \mathtt{TC}\ \overline{u}) \hookrightarrow \overline{g_i\ (fv(\overline{u}^{i_n})) : F_{\mathtt{TC}_i}(\overline{u}^{i_n}) \sim \theta_i(u_{i_0})}^m} \ \textsc{AxGen}$$

Premise $fv(\theta_i(u_{i_0})) \subseteq fv(\overline{u}^{i_n})$ ensures that the generated axioms are well-formed; like the Liberal Coverage Condition (Sulzmann et al., 2007b) it checks that the image of every functional dependency is determined by its domain.

Method Translation & Type Subsumption Since method implementations are in effect explicitly typed, we need a procedure for deciding *type subsumption*. We say that a polytype σ_1 *subsumes* polytype σ_2, if any expression that can be assigned type σ_1 can also be assigned type σ_2. Since we elaborate during inference, we perform type inference and the subsumption check simultaneously, by means of relation $P; \Gamma \vdash_{\mathtt{TM}} e : \sigma \rightsquigarrow t$, which is given by rule:

$$\dfrac{\begin{array}{c} \Gamma \vdash_{\mathtt{TM}} e : \tau_1 \rightsquigarrow t \mid \Pi; \mathcal{E} \qquad \Gamma \vdash_{\mathtt{TY}} (\forall \bar{a}.\, \overline{\pi} \Rightarrow \tau_2) \qquad \overline{\Gamma \vdash_{\mathtt{CC}} \pi \rightsquigarrow \tau} \\ \omega, \bar{d} \text{ fresh} \qquad (\overline{d : \pi}) \Downarrow \mathbb{E} \qquad P, (\overline{d : \pi}), P_{\mathbb{E}} \vdash_{\mathtt{E}} \Pi, \mathcal{E}, (\omega : \tau_1 \sim \tau_2) \rightsquigarrow^! \bullet; \theta; \eta \end{array}}{P; \Gamma \vdash_{\mathtt{TM}} e : (\forall \bar{a}.\, \overline{\pi} \Rightarrow \tau_2) \rightsquigarrow \Lambda \bar{a}.\, \lambda \overline{(d : \tau)}.\, \mathbb{E}[\eta(\theta(t \triangleright \omega))]} \ (\preceq)$$

In short, from the assumption $\overline{\pi}$ we need to be able to completely derive all constraints that arise from typing e and the equality $(\tau_1 \sim \tau_2)$. We locally extend the program theory with the transitive closure of the superclass relation on $\overline{\pi}$, thus exposing both superclass dictionaries and FD constraints induced by $\overline{\pi}$.

Superclass Entailment Furthermore, we need to ensure that the instance context $\overline{\pi}$ (along with the newly created axioms P_{ax}) completely entails the superclass and FD constraints. This procedure is captured by relation $P_{inst} \vdash_{\texttt{SC}} (\texttt{TC}\ \overline{u}) \rightsquigarrow (\overline{\tau}, \overline{t}, \overline{\gamma})$:

$$\frac{\textbf{class}\ \forall \overline{a}\overline{b}.\ \overline{\pi} \Rightarrow \texttt{TC}\ \overline{a} \mid \overline{fd}^{m} \qquad \overline{\omega}, \overline{d}\ \text{fresh} \qquad \theta = [\overline{u}/\overline{a}] \cdot \mathit{det}(\overline{a}, \overline{\pi}) \\ P_{inst} \vdash_{\texttt{E}} (\overline{d : \theta(\pi)}), (\overline{\omega : \theta(F_{\texttt{TC}_i}(\overline{a}^{i_n}) \sim a_{i_0})}) \rightsquigarrow^{!} \bullet; \theta_s; \eta_s}{P_{inst} \vdash_{\texttt{SC}} (\texttt{TC}\ \overline{u}) \rightsquigarrow (\theta_s(\theta(\overline{b})), \eta_s(\overline{d}), \eta_s(\overline{\omega}))}\ \text{SC}$$

Notice that the relation also computes the existential types introduced in the superclass context $\theta_s(\theta(\overline{b}))$, which should also be stored in the resulting GADT dictionary.

Instance Elaboration Finally, Rule INSTANCE utilizes the above relations to produce the dictionary transformer d_I, which reflects the *Instance Constraint Scheme* (Equation CS2a).

Since we do not encode the superclass relation using constraint schemes but via match contexts, both the method elaboration and the superclass entailment are performed under environment P_{all}, which includes not only the instance context $\overline{\pi}$ and axioms P_{ax}, but also the transitive closure of the superclass relation P', obtained by exhaustively destructing assumptions $\overline{\pi}$.

8.6 Meta-theory

This section considers the key meta-theoretical properties of both the type system and the type inference and elaboration algorithm.

8.6.1 Termination of Type Inference

First, we consider termination of type inference. In order to ensure that the algorithm of Section 8.5 is terminating, we extend the termination conditions of Section 6.5 accordingly:

(a) The superclass relation forms a *directed acyclic graph* (DAG).

(b) In each class instance ($\textbf{instance}\ \forall \overline{a}\overline{b}.\ \overline{\pi} \Rightarrow \texttt{TC}\ \overline{u}$):

- no variable has more occurrences in a type class constraint π than the head ($\texttt{TC}\ \overline{u}$), and

- each class constraint π in the context $\overline{\pi}$ has fewer constructors and variables (taken together, counting repetitions) than the head ($\texttt{TC}\ \overline{u}$).

(c) For every generated axiom $(g\ \overline{a} : F(\overline{u}) \sim \tau)$, in every subterm $(F_1(\overline{\tau}_1) \subseteq \tau)$:

- there is no subterm $(F_2(\overline{\tau}_2) \subseteq F_1(\overline{\tau}_1))$,
- the sum of the number of type constructors and type variables is smaller than the corresponding number in $\overline{u}$, and
- there are not more occurrences of any variable a than in $\overline{u}$.

In essence, restrictions (a) and (b) are exactly the same as for the basic system, but they are extended to account for multi-parameter type classes.

The first restriction ensures that relations $det(\overline{a}, \overline{\pi})$ and $(\Pi \Downarrow \mathbb{E})$ terminate, since they both compute the transitive closure of the superclass relation.

The second restriction ensures that instance contexts are *decreasing*, so that class resolution (Rules $\textsc{Cls}_E$ and Rules $\textsc{Red}_E$) is also terminating, given that the type equality axioms are strongly normalizing.

Lastly, the third restriction (borrowed from Schrijvers et al. (2008, Def. 5)) ensures that the generated axioms are strongly normalizing, that is, confluent and terminating, which allows us to turn constraint entailment (Section 8.5.5) into a deterministic function.

Conjecture 1 (Termination of Type Inference)**.** *If a program satisfies the Termination Conditions, then type inference terminates.*

8.6.2 Functional Dependency Property

There are two important properties that regulate the functional dependency property.

Compatibility Firstly, we need to make sure that there are no two conflicting definitions that associate two different values with the same key. To this end, we impose the *Compatibility Condition*:

Definition 9 (Compatibility Condition)**.** *Let there be a class declaration and any pair of instance declarations for that class:*

$$\begin{array}{l}
\mathbf{class}\ \forall \overline{a}\overline{b}.\ \overline{\pi} \Rightarrow \texttt{TC}\ \overline{a} \mid fd_1, \ldots, fd_m\ \mathbf{where}\ f :: \sigma \\
\mathbf{instance}\ \forall \overline{a}_1\overline{b}_1.\ \overline{\pi}_1 \Rightarrow \texttt{TC}\ \overline{u}_1\ \mathbf{where}\ f = e_1 \\
\mathbf{instance}\ \forall \overline{a}_2\overline{b}_2.\ \overline{\pi}_2 \Rightarrow \texttt{TC}\ \overline{u}_2\ \mathbf{where}\ f = e_2
\end{array}$$

Then, for each functional dependency $fd_i \equiv a_{i_1}, \ldots, a_{i_n} \to a_{i_0}$ *the following should hold:*

$$compat(F_{\texttt{TC}_i}(\overline{u}_1^{i_n}) \sim \theta_{i1}(u_{i_0 1}), F_{\texttt{TC}_i}(\overline{u}_2^{i_n}) \sim \theta_{i2}(u_{i_0 2}))$$

where $\theta_{i1} = det(fv(\overline{u}_1^{i_n}), \overline{\pi}_1)$ *and* $\theta_{i2} = det(fv(\overline{u}_2^{i_n}), \overline{\pi}_2)$*.*

Relation $compat(\cdot, \cdot)$ is the *compatibility relation*, as defined by Eisenberg et al. (2014):

Definition 10 (Compatibility)**.** *Two equalities* $\phi_1 = F(\overline{u}_1) \sim \tau_1$ *and* $\phi_2 = F(\overline{u}_1) \sim \tau_2$ *are compatible—denoted as* $compat(\phi_1, \phi_2)$*—iff* $unify(\overline{u}_1; \overline{u}_2) = \theta$ *implies* $\theta(\tau_1) = \theta(\tau_2)$*.*

In the nomenclature of Jones (2000, Section 6.1) and Sulzmann et al. (2007b, Def. 6–8) compatibility is known as *consistency*. Both works impose a very conservative consistency condition, which requires, for any two instance heads ($\texttt{TC}\ \overline{u}_1$) and ($\texttt{TC}\ \overline{u}_2$) and any functional dependency $fd_i \equiv \overline{a}^{i_n} \to a_{i_0}$ of class TC, that $\theta(u_{i_0 1}) = \theta(u_{i_0 2})$ if $unify(\overline{u}_1^{i_n}; \overline{u}_2^{i_n}) = \theta$. This means that the function is fully determined by the instance head, and cannot depend on the instance context. The latter is supported by our more liberal Compatibility Condition, which meets Challenge 1 of Section 8.1.3, by providing a criterion to verify the consistency of more liberal instances.

Notice that both Jones and Sulzmann et al. consider an additional property, *coverage*, which stipulates that the image of every functional dependency instance is fully determined by its domain. Jones enforces this property through a conservative condition, while Sulzmann et al. consider the more liberal *Liberal Coverage Condition* (Sulzmann et al., 2007b, Def. 15) which also takes the instance context into account. Our system does not require an external coverage condition as it already internalizes coverage in the determinacy relation (Section 8.3.5).

Unambiguous Witness Functions Secondly, we need to make sure that the function that witnesses the functional dependency is uniquely determined. For this reason, we impose the *Unambiguous Witness Condition*.

Definition 11 (Unambiguous Witness Condition)**.** *Let there be a class declaration and any instance for that class:*

$$\begin{array}{l}\textbf{class}\ \forall \overline{a}\overline{b}.\ \overline{\pi} \Rightarrow \texttt{TC}\ \overline{a} \mid \overline{fd}^m\ \textbf{where}\ f :: \sigma \\ \textbf{instance}\ \forall \overline{a}'\overline{b}'.\ \overline{\pi}' \Rightarrow \texttt{TC}\ \overline{u}\ \textbf{where}\ f = e\end{array}$$

Then, for each functional dependency $fd_i \equiv \overline{a}^{i_n} \to a_{i_0}$ *it is required that* $det(fv(\overline{u}^{i_n}), \overline{\pi}')$ *is unambiguous on* $fv(u_{i_0})$*.*

A witness derivation $det(\overline{a}, \overline{\pi}) = \theta$ is unambiguous *on type variables* $\overline{b}$ iff $\overline{b} \subseteq dom(\theta)$ and $\theta(\overline{b})$ is independent of the order in which relation $\overline{a}; \overline{\pi} \vdash_{\mathtt{D}} \theta_1 \rightsquigarrow \theta_2$ selects class constraints π from $\overline{\pi}$.

To see why this condition is important, consider for example the following declarations:

$$\begin{array}{ll} \textbf{class } C_1\ a\ b \mid a \to b & \qquad \textbf{class } C\ a\ b \mid a \to b \\ \textbf{class } C_2\ a\ b \mid a \to b & \qquad \textbf{instance } (C_1\ a\ b, C_2\ a\ b) \Rightarrow C\ [a]\ [b] \end{array}$$

What axiom should the C instance give rise to? Using the instance context $(C_1\ a\ b, C_2\ a\ b)$, we can derive either of the two:

$$\begin{array}{l} \textbf{axiom } g_1\ a : F_C\ a \sim [F_{C_1}\ a] \\ \textbf{axiom } g_2\ a : F_C\ a \sim [F_{C_2}\ a] \end{array}$$

Yet, depending on the choice, different programs are accepted. For example, from the given constraints $\{C_1\ a\ b, C_2\ a\ c, C\ [a]\ [b]\}$ we can derive $(b \sim c)$ if g_1 is available; the same does not hold for g_2.

Even worse, the choice of the axiom affects the compatibility of the instance with other instances for the same class. To support modular compilation, we cannot optimize the choice by taking the rest of the program into account.

For these reasons, our Unambiguous Witness Condition rejects programs with such ambiguity. Another solution would be for the programmer to manually resolve the ambiguity by expressing a preference.

8.6.3 Type Substitution Property

We believe that our system satisfies the type substitution property.

Conjecture 2. *If* $I; \Gamma \vdash_{\mathtt{TM}} e : \forall a.\ \sigma$ *and* $\Gamma \vdash_{\mathtt{TY}} \tau$, *then* $I; \Gamma \vdash_{\mathtt{TM}} e : [\tau/a]\sigma$.

In short, a type system satisfies the type substitution property iff typing a term e with type $\forall a.\ \sigma$ implies that we can also type it with the instantiated type $[\tau/a]\sigma$.

Chakravarty et al. (2005a) used the following example to compare three systems (the system implemented by the Hugs compiler, the system implemented by GHC, and a system designed by Stuckey and Sulzmann (2005)) with respect to whether they satisfy this property.

$$\begin{array}{l} \textbf{class } C\ a\ b \mid a \to b\ \textbf{where } \{\ foo :: a \to b\ \} \\ \textbf{instance } C\ Bool\ Int\ \textbf{where } \{\ foo = \ldots\ \} \end{array}$$

Three possible signatures for function *bar* are:

$$\begin{array}{llll} bar :: C\ a & b \Rightarrow a & \rightarrow b & \text{(1: most general type)} \\ bar :: C\ Bool & b \Rightarrow Bool & \rightarrow b & \text{(2: substitution instance)} \\ bar :: & & Bool \rightarrow Int & \text{(3: apply the fd)} \\ bar = foo & & & \end{array}$$

All three signatures are accepted by the system of Stuckey and Sulzmann, which is based on Constraint Handling Rules. Yet, signature (2) is rejected by both GHC and Hugs, which use a dictionary-based translation to an explicitly-typed language based on System F. Our system accepts all three signatures.

As far as we know, our system is the first with functional dependencies to satisfy the type substitution property, while translating to a typed intermediate language. In the future, we would like to formally verify this expectation.

8.6.4 Algorithm Soundness

"The worst form of inequality is to try to make unequal things equal."

—Aristotle

The algorithm performs two tasks at once: type inference and elaboration into System F_C. We conjecture that it is sound on both accounts.

Firstly, the type inference task is sound.

Conjecture 3 (Soundness of Type Inference)**.** *If* $\vdash_{\texttt{PGM}} pgm \rightsquigarrow \overline{decl}$, *then* $\vdash_{\texttt{PGM}} pgm$.

Secondly, the elaboration produces well-typed System F_C code.

Conjecture 4 (Preservation of Typeability Under Elaboration)**.** *If* $\vdash_{\texttt{PGM}} pgm \rightsquigarrow \overline{decl}$, *then* $\vdash_{\texttt{PGM}} \overline{decl}$.

Moreover, to be type safe, System F_C requires the consistency of the axiomatic equational theory. This property follows from the Compatibility Condition:

Theorem 14 (Consistency of Elaborated Programs)**.** *If pgm satisfies the Compatibility Condition and* $\vdash_{\texttt{PGM}} pgm \rightsquigarrow \overline{decl}$, *then the top-level typing environment of* $\overline{decl}$ *is consistent (according to the definition of System* F_C *consistency (Sulzmann et al., 2007a)).*

8.6.5 Ambiguity

Following the Haskell tradition, we also require that *ambiguous* type signatures are rejected, since the run-time behavior of terms that inhabit them is not well-specified. Checking signatures for ambiguity is straightforward:

Conjecture 5 (Non-ambiguous Types)**.** *Let there be a (well-scoped) type* $\sigma = \forall \overline{a}.\ \overline{\pi} \Rightarrow \tau$*. If* $fv(\overline{\pi}) \subseteq dom(det(fixed(\tau), \overline{\pi})) \cup fixed(\tau)$*, then* σ *is unambiguous.*

Function $fixed(\cdot)$ computes the set of *fixed* variables of a monotype:

$$\begin{array}{lcllcl} fixed(a) & = & \{a\} & \qquad fixed(\tau_1\ \tau_2) & = & fixed(\tau_1) \cup fixed(\tau_2) \\ fixed(T) & = & \varnothing & \qquad fixed(F(\overline{\tau})) & = & \varnothing \end{array}$$

Intuitively, all type variables appearing in the context $\overline{\pi}$ should be determined from the monotype τ, either by directly appearing in τ, or indirectly via a functional dependency (or a chain of them). For instance, given the class declaration **class** $C\ a\ b \mid a \rightarrow b$ we conclude that signature $C\ a\ b \Rightarrow a \rightarrow a$ is unambiguous because type variable b is functionally determined by a.

8.6.6 Principality of Types

Furthermore, we conjecture that the specification of Section 8.3 has the *principal type property*:

Conjecture 6 (Principal Types)**.** *If* e *is well-typed, then there exists a type* σ_0 *(the principal type), such that* $I; \Gamma \vdash_{\texttt{TM}} e : \sigma_0$ *and, for all* σ *such that* $I; \Gamma \vdash_{\texttt{TM}} e : \sigma$*, we have that* $I; \Gamma \models \sigma_0 \preceq \sigma$*.*

Here relation $I; \Gamma \models \sigma_0 \preceq \sigma$ defines *type subsumption*:

$$\frac{I, [\overline{\tau}/\overline{b}]\overline{\pi}_2; \Gamma \models \overline{\pi}_1, \tau_1 \sim [\overline{\tau}/\overline{b}]\tau_2 \qquad \Gamma \vdash_{\texttt{TY}} \overline{\tau}}{I; \Gamma \models (\forall \overline{a}.\ \overline{\pi}_1 \Rightarrow \tau_1) \preceq (\forall \overline{b}.\ \overline{\pi}_2 \Rightarrow \tau_2)}\ (\preceq)$$

Moreover, without introducing further formal notation, we state our expectation that type inference derives the principal type:

Conjecture 7 (Inference Computes Principal Types)**.** *The type inference of Section 8.5 computes only principal types.*

8.6.7 Coherence

To ensure that elaboration is coherent in the presence of type classes with functional dependencies, we require the same conditions as for the basic system:

non-overlapping instances (Definition 3), and non-ambiguous types (Definition 4). We generalize the first condition to account for multiple class parameters:

Definition 12 (Non-overlapping Instances)**.** *Any two instance heads* $(TC\,\overline{u}_1)$ *and* $(TC\,\overline{u}_2)$ *for the same class should not overlap* ($\nexists\theta.\ \theta(\overline{u}_1) = \theta(\overline{u}_2)$)*.*

The second condition should also be generalized, to account for existentially-quantified type variables that can be introduced by functional dependencies. Hence, instead of the simple unambiguous condition for type classes (predicate $\mathit{unamb}(\cdot)$ in Definition 4), in our definition of non-ambiguous types we require the generalized condition $\mathit{unambig}(\overline{b}, \overline{a}, \overline{\pi})$ (see Section 8.3.4). Since this condition is already enforced by the specification of Section 8.3, the only external condition we require is Definition 12.

8.6.8 Completeness

Finally, we conjecture that our algorithm is complete with respect to the declarative type system for programs that satisfy the Termination and Unambiguous Witness Conditions.

8.7 Related Work

Functional Dependencies Functional dependencies were introduced in Haskell's class system by Jones (2000), and its first sound and decidable type inference has been given by Duck et al. (2004). Follow-up work by Sulzmann et al. (2007b) formalized functional dependencies in terms of *Constraint Handling Rules* (Frühwirth, 1995), and thoroughly studied several extensions, including multi-range functional dependencies and weakened variants of the coverage condition.

Non-Functional Improvement Our work treats functional dependencies as type-level functions, as originally intended by Jones (2000). Alternatively, one can view functional dependencies as a more general mechanism for guiding type inference, e.g., as asked for in GHC feature request #8634. Under this interpretation, the domain of a functional dependency does not necessarily determine its image; the result can also be partially determined. This fits with Jones' more general theory of *improvement* (Jones, 1995c). A flexible system for improvement was presented by Stuckey and Sulzmann (2005), where the programmer can extend type inference, including partial improvements, directly through Constraint Handling Rules. It is an open question how to integrate this approach with a typed intermediate language.

Type Families Functional Dependencies are most closely related to associated type synonyms (Chakravarty et al., 2005a; Schrijvers et al., 2007, 2008) and our formalization is based on theirs. This enables a more direct comparison and integration of both features in future work.

Our Compatibility Condition is based on that of Eisenberg et al. (2014) for open type families, which relaxes the non-overlapping check of Schrijvers et al. (2008).

Injective Associated Type Synonyms Stolarek et al. (2015) proposed injectivity annotations for type families, which closely resemble functional dependencies in both syntax and semantics: to indicate that a type family with result type b is injective in the i-th argument a_i a user can add an annotation $b \rightarrow a_i$. They introduce a new System $\mathrm{F_C}$ coercion form to witness injectivity. Our elaboration does not need this new form, as the type class dictionary serves to hold the witness; this approach could be used for the injectivity of associated type synonyms too.

Expanding on an earlier sketch by Schrijvers and Sulzmann (2008), Serrano et al. (2015) discuss an approach for elaborating type classes into type families augmented with dictionaries. They use injectivity annotations to elaborate functional dependencies.

8.8 Scientific Output

The material found in this chapter is drawn from the following publication:

> Georgios Karachalias and Tom Schrijvers (2017). Elaboration on Functional Dependencies: Functional Dependencies Are Dead, Long Live Functional Dependencies! In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell '17, pp. 133–147, Oxford, UK, September 7–8, 2017.

This work has tackled a number of important open challenges concerning functional dependencies. Firstly, we summarize the shortcoming in the treatment of functional dependencies. We address these shortcomings by providing a formalization of functional dependencies that explicitly reconstructs the implicit type-level function that witnesses each functional dependency. Alongside the declarative type system, we develop a type inference algorithm with evidence translation from source terms to System F_C that is faithful to our type system specification. Furthermore, our algorithm uses the same building blocks as that of associated type synonyms, which makes us confident that our work enables the proper integration of functional dependencies in Haskell's eco-system of advanced type-level features, an aspiration we return to in Section 10.3.2. Lastly, our work takes a significant step towards answering a long-standing question of whether functional dependencies and (associated) type families have the same expressive power: by elaborating functional dependencies into type families and type equalities, we essentially show that the latter subsume the former. Proving (or disproving) the other direction remains an open question to be answered in the future.

Chapter 9

Bidirectional Instances

Finally, in this chapter we present the third and last type class extension we have developed: *Bidirectional Instances*.

The chapter is structured as follows: Section 9.1 motivates the development of bidirectional instances and presents their intended semantics. Section 9.2 discusses the challenges that arose during the development of the feature, as well as the design choices we have made to address them. Section 9.3 formalizes the differences between the specification of the basic system we presented in Section 6.3 and a system extended with bidirectional instances, as well as the changes bidirectional instances introduce to the inference (and elaboration) algorithm of the basic system (Section 6.4). Finally, Section 9.4 covers the meta-theoretical properties of bidirectional instances and Section 9.5 summarizes the main outcomes of this work.

9.1 Motivation

9.1.1 Structural Induction Over Indexed Data Types

Ever since GADTs were introduced in Haskell (Peyton Jones et al., 2006), they have been put to good use by programmers for dataflow analysis and optimization (Ramsey et al., 2010), accelerated array processing,[1] automatic

[1] `https://hackage.haskell.org/package/accelerate`

differentiation,[2] and more beyond.[3] Yet, their interaction with existing features such as type classes (Wadler and Blott, 1989) and functional dependencies (Jones, 2000) has a lot of room for improvement.

For example, consider (a simplified version of) the *Term* datatype, as given by Johann and Ghani (2008):

$$\begin{array}{l} \textbf{data}\ \mathit{Term} :: \star \to \star\ \textbf{where} \\ \quad \mathit{Con}\ ::\ a \to \mathit{Term}\ a \\ \quad \mathit{Tup}\ ::\ \mathit{Term}\ a \to \mathit{Term}\ b \to \mathit{Term}\ (a, b) \end{array}$$

Datatype *Term* encodes a simple expression language, with constants (constructed by data constructor *Con*) and tuples (constructed by data constructor *Tup*).

Making (*Term a*) an instance of even the simplest of type classes can be challenging. For example, the following straightforward instance is not typeable under the current specification of type classes:

$$\begin{array}{l} \textbf{instance}\ \mathit{Show}\ a \Rightarrow \mathit{Show}\ (\mathit{Term}\ a)\ \textbf{where} \\ \quad \mathit{show}\ \ (\mathit{Con}\ x) \quad\ = \mathit{show}\ x \\ \quad \mathit{show}\ \ (\mathit{Tup}\ x\ y) = \mathit{unwords}\ [\texttt{"("}, \mathit{show}\ x, \texttt{","}, \mathit{show}\ y, \texttt{")"}] \end{array}$$

Loading the above program into `ghci` emits the following error(s):

```
Bidirectional.hs:14:33:
    Could not deduce (Show b) arising from a use of ‘show’
    from the context (Show a) or from (a ~ (b, c))

Bidirectional.hs:14:44:
    Could not deduce (Show c) arising from a use of ‘show’
    from the context (Show a) or from (a ~ (b, c))
```

As the message indicates, the source of the errors is the recursive calls to *show* in the second clause: the instance context (*Show a*) and the local constraint (exposed via GADT pattern matching) $a \sim (b, c)$ are not sufficient to prove (*Show b*) and (*Show c*), as is required by the recursive calls to *show*. In summary, the type system cannot derive the following implications:

$$\begin{array}{l} \forall b.\ \forall c.\ \mathit{Show}\ (b, c) \ \Rightarrow\ \mathit{Show}\ b \\ \forall b.\ \forall c.\ \mathit{Show}\ (b, c) \ \Rightarrow\ \mathit{Show}\ c \end{array}$$

[2]`https://hackage.haskell.org/package/ad`

[3]Fun fact: despite the current popularity of GADTs, they were first introduced in an unpublished draft by Augustsson and Petersson (1994), under the name of "silly type families".

The Problem Both implications above constitute the inversion of the implication derived by the predefined *Show* instance for tuples:

$$\textbf{instance}\ (\mathit{Show}\ b, \mathit{Show}\ c) \Rightarrow \mathit{Show}\ (b, c)\ \textbf{where}\ \{\ \dots\ \}$$

Indeed, as we discussed earlier in Sections 5.4.2 and 5.4.3, both the logical and the constructive interpretation of type classes in most[4] existing system is not *bidirectional*: the system can only derive *Show* (b, c) from $(\mathit{Show}\ b, \mathit{Show}\ c)$, but not the other way around.

9.1.2 Functional Dependencies and Associated Type Families

Unfortunately, the lack of bidirectionality of type class instances does not affect only the expressive power of simple type classes, but also the expressive power of a multitude of features based on them, such as functional dependencies (Jones, 2000) and associated type families (Chakravarty et al., 2005a).

For example, let us consider an example of type-level programming using functional dependencies.[5] Recall the definition of type-level natural numbers and length-indexed vectors we gave in Section 2.3:

```
data Nat :: ⋆ where                 data Vec :: Nat → ⋆ → ⋆ where
  Zero :: Nat                         VN :: Vec Zero a
  Succ :: Nat → Nat                   VC :: a → Vec n a → Vec (Succ n) a
```

Equipped with type-level natural numbers, we can encode type-level addition (using the Peano axioms (Peano, 1889)) by means of a multi-parameter type class and functional dependencies:

```
class Add (n :: Nat) (m :: Nat) (k :: Nat) | n m → k
instance                Add  Zero      m  m
instance Add n m k  ⇒  Add  (Succ n)  m  (Succ k)
```

parameters n and m represent the operands, and parameter k represents the result, which is uniquely determined by the choice of n and m. The two Peano axioms for addition correspond to two instances for class *Add*, one for each form n can take.

[4]Sulzmann et al. (2007b) interpret class instances bidirectionally, but the CHR-based interpretation of type classes does not always align with the dictionary-passing elaboration we are targeting here, as we discussed in Chapter 8.

[5]A similar example has been presented by Hallgren (2000), who implemented insertion sort at the level of types using functional dependencies.

All the above can be combined to define function *append*, which concatenates two length-indexed vectors:

$$\begin{array}{lllll} \mathit{append} :: \mathit{Add}\ n\ m\ k \Rightarrow \mathit{Vec}\ n\ a \to \mathit{Vec}\ m\ a \to \mathit{Vec}\ k\ a \\ \mathit{append}\ \ \mathit{VN} & \mathit{ys} & = & \mathit{ys} \\ \mathit{append}\ \ (\mathit{VC}\ x\ \mathit{xs}) & \mathit{ys} & = & \mathit{VC}\ x\ (\mathit{append}\ \mathit{xs}\ \mathit{ys}) \end{array}$$

The implementation of *append* is identical to the corresponding one for simple lists but its signature is much richer: *append* takes two vectors of length n and m, and computes a vector of length k, where $n{+}m = k$. Types like the above are extremely useful for example in linear algebra libraries (see for example Hackage package `linear`), to ensure that operations respect the expected dimensions.

Unfortunately, the above example fails to type-check, due to the lack of an evidence-based translation of functional dependencies. Yet, even with the development of functional dependencies we presented in Chapter 8 the above program is ill-typed.

Once again, the key element missing is bidirectional instances. In the second clause of *append*, the recursive invocation of *append* requires $(\mathit{Add}\ n'\ m\ k')$, while the signature provides $(\mathit{Add}\ (\mathit{Succ}\ n')\ m\ (\mathit{Succ}\ k'))$.[6] That is, we need the following implication:

$$\forall n'.\ \forall m.\ \forall k'.\ \mathit{Add}\ (\mathit{Succ}\ n')\ m\ (\mathit{Succ}\ k') \Rightarrow \mathit{Add}\ n'\ m\ k'$$

which can be obtained by interpreting the second *Add* instance bidirectionally.

As we argued in the previous chapter, associated type families (Chakravarty et al., 2005a) share—for the most part—their semantics with functional dependencies. Thus, the problem we are presenting here applies to associated type families as well; shortcomings of type classes affect all their extensions.

In summary, the lack of bidirectionality of type class instances severely reduces the expressive power of type class extensions, such as associated types (Chakravarty et al., 2005b), associated type synonyms (Chakravarty et al., 2005a), and functional dependencies (Jones, 2000).

Workaround One might argue that the problem can be avoided by using *Open Type Families* (Schrijvers et al., 2007, 2008). Indeed, we can implement addition

[6] In fact, the signature provides $(\mathit{Add}\ n\ m\ k)$, which we can refine using $n \sim \mathit{Succ}\ n'$ (obtained by GADT pattern matching), and the type-level function introduced by the functional dependency (see Chapter 8).

by means of an open type family *Add*:

$$\begin{array}{llll}\textbf{type family}\ \mathit{Add}\ (n :: \mathit{Nat})\ (m :: \mathit{Nat}) :: \mathit{Nat} & & & \\ \textbf{type instance}\ \ \mathit{Add}\ \ \mathit{Zero} & m & = & m \\ \textbf{type instance}\ \ \mathit{Add}\ \ (\mathit{Succ}\ n) & m & = & \mathit{Succ}\ (\mathit{Add}\ n\ m)\end{array}$$

and assign *append* the alternative signature:

$$\mathit{append} :: \mathit{Vec}\ n\ a \to \mathit{Vec}\ m\ a \to \mathit{Vec}\ (\mathit{Add}\ n\ m)\ a$$

Nevertheless, this workaround is not a panacea. First, it does not address the original problem of type classes being insufficiently expressive. Second, as Morris and Eisenberg (2017) have shown, open type families severely complicate the meta-theory of the system, and should thus be avoided. One of the most important problems with open type families is partiality, which requires infinite unification, while Haskell does not accept infinite types. This issue constitutes the main motivation for the development of the work of Morris and Eisenberg (2017). We refer the reader to their work for more details on this debate.

9.1.3 Constrained Type Families

Morris and Eisenberg (2017) recently provided compelling arguments for the replacement of open type families with what they call *Constrained Type Families*. Constrained type families, similarly to associated type families, use the generic notion of qualified types (Jones, 1992) to capture the domain of a type family within a predicate.

Within this setting, the bidirectionality of the axioms is essential. Indeed, Morris and Eisenberg use the *append* example above to motivate the extension of System F_C with the `assume` construct, which axiomatically provides the bidirectionality needed for *append* to type check.

9.1.4 Summary

In summary, the lack of a bidirectional elaboration of class instances seriously undermines the interaction between GADTs and type classes, as well as type class extensions. In the remainder of this chapter we present a solution to this problem, which shows how to elaborate class instances bidirectionally in System F_C, without additional extensions.

9.2 Technical Challenges

Though bidirectional instances are sorely needed for applications involving GADTs, the problem is more general. For example, GHC currently rejects programs where wanted constraint $Eq\ a$ needs to be derived from given constraint $Eq\ [a]$. This is the case for the following type-annotated function:

$$\begin{array}{l} cmp :: Eq\ [a] \Rightarrow a \rightarrow a \rightarrow Bool \\ cmp\ x\ y = x \texttt{ == } y \end{array}$$

Though contrived, function *cmp* is a minimal example that exhibits all problems that arise in elaborating class instances bidirectionally in the well-established dictionary-passing translation (Hall et al., 1996). Thus, we use it as our running example throughout the remainder of this section to discuss all the technical challenges of interpreting type class instances bidirectionally.

9.2.1 Key Idea

Why Are Instances Bidirectional The first question one might ask is *why* can the instance axiom be inverted; why can type class instances be interpreted bidirectionally.

Existing systems with type classes ensure *coherence* by disallowing instance heads to overlap (Definition 3). If no instance heads overlap, then the entailment of a class constraint can have at most one derivation tree if no local constraints are present (if local constraints are available (e.g., from a type signature), multiple derivation trees might exist but they all have the same computational content; only class instances provide the computational content). Thus, the evidence term (dictionary) for any constraint ($\texttt{TC}\ \tau$) is uniquely determined by the parameter τ. For example, given that instances are non-overlapping, the only way one can derive $Eq\ [Int]$ is by using the following two instances:

$$\begin{array}{l} \textbf{instance}\ Eq\ Int \\ \textbf{instance}\ Eq\ a \Rightarrow Eq\ [a] \end{array}$$

One could also derive $Eq\ [Int]$ from a given constraint $Ord\ [Int]$, since Eq is a superclass of Ord. Yet, computational content is only given by instance declarations; the dictionary of type $T_{Eq}\ [Int]$ stored within the dictionary of type $T_{Ord}\ [Int]$ *has to* be computed using the above two instances.

That being said, the only way one can create a dictionary of type $Eq\ [a]$, for any type a, is by using the Eq instance for lists. Consequently, if a constraint $Eq\ [a]$ is *given*, one can safely assume that $Eq\ a$ is also available: modus ponens is invertible if there is no overlap in the implication heads.

General Strategy Unfortunately, bidirectionality of class instances is a meta-property of type classes: the system does not utilize it to become more powerful. Additionally, it is a fragile property: if we allow instances to overlap, the property no longer holds (if there are multiple ways to derive the same constraint, one can no longer invert the instance axiom).

In order to integrate the property within the system, we need to show how to derive the instance context from the instance head *constructively*. To achieve this, our approach is simple: *reuse the infrastructure of superclasses.*

Superclass dictionaries are stored within subclass dictionaries. Hence, a superclass constraint (e.g., $Eq\ a$) can always be derived from a subclass constraint (e.g., $Ord\ a$), which is constructively reflected in a System F projection function. Thus, our key idea is to store the instance context within the class dictionary and retrieve it when necessary using System F projection functions.

This technique poses several technical challenges, which we elaborate on in the remainder of this section.

9.2.2 Challenge 1: Lack of Parametricity

Possibly the biggest challenge in interpreting class instances bidirectionally lies in the non-parametric dictionary representation. Let us consider the standard equality class Eq, along with 3 instances:

$$\textbf{class}\ Eq\ a\ \textbf{where}\ \{\ (==) :: a \to a \to Bool\ \}$$

$$\begin{array}{llllr}
\textbf{instance} & & Eq\ Int & \textbf{where}\ \{\ (==) = \ldots\ \} & \texttt{(I1)} \\
\textbf{instance}\ Eq\ b & \Rightarrow & Eq\ [b] & \textbf{where}\ \{\ (==) = \ldots\ \} & \texttt{(I2)} \\
\textbf{instance}\ (Eq\ c, Eq\ d) & \Rightarrow & Eq\ (c, d) & \textbf{where}\ \{\ (==) = \ldots\ \} & \texttt{(I3)}
\end{array}$$

The instance context for each instance varies, depending on the instance parameter a: instance `(I1)` has an empty context, instance `(I2)` has context $Eq\ a$, and instance `(I3)` has context $(Eq\ a, Eq\ b)$. In the well-established dictionary-passing elaboration approach (Wadler and Blott, 1989; Hall et al., 1996), these contexts correspond to the following System F types:

$$\begin{array}{lll}
\texttt{(I1)} : & () & , \text{where } a \sim Int \\
\texttt{(I2)} : & T_{Eq}\ b & , \text{where } a \sim [b] \\
\texttt{(I3)} : & (T_{Eq}\ c, T_{Eq}\ d) & , \text{where } a \sim (c, d)
\end{array}$$

where T_{Eq} is the System F type constructor for the class dictionary. Depending on how we refine the class parameter a, the representation of the instance context in System F can be different.

9.2.3 Challenge 2: Termination of Type Inference

Typing Specification Though elaboration exhibits a non-parametric behavior, in terms of logic (and typing) bidirectional instances can be expressed via universal quantification alone. For example, the above three instances can be interpreted as the following formulas:[7]

$$\begin{array}{lll} & Eq\ Int & \leftrightarrow\ True \\ \forall a. & Eq\ [a] & \leftrightarrow\ Eq\ a \\ \forall b.\ \forall c. & Eq\ (b,c) & \leftrightarrow\ (Eq\ b \wedge Eq\ c) \end{array}$$

Of course, the one direction of the implication is already derived by the specification of the basic system; for extending the system with bidirectional instances one would simply have to add implications for the other direction.

Type Inference and Termination The specification of typing is not affected much by bidirectional instances but this is not the case for type inference. Consider for example the inversion of the *Eq* $[a]$ instance:

$$\forall a.\ Eq\ [a] \rightarrow Eq\ a$$

If such axioms are not used with care, the termination of the type inference algorithm is threatened. The standard backwards-chaining entailment (see Section 6.3) cannot use such axioms to simplify goals and terminate. For example, we can "simplify" $Eq\ \tau$ to $Eq\ [\tau]$ using the above axiom. The size of the "simpler" constraint $Eq\ [\tau]$ is bigger than the one we started with. What is more, the axiom can be applied infinitely many times (to capture that all nested list types are instances of *Eq*): the resolution tree now contains infinite paths. Thus, even the backtracking approach we showed in Chapter 7 cannot handle bidirectional instances in an obvious way: bidirectional axioms need to be used selectively to ensure the termination of type inference.

9.2.4 Challenge 3: Principality of Types

Finally, the introduction of bidirectional instances threatens the principality of types. In the absence of bidirectional instances, function *cmp* would have a single most general type:

$$cmp :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

Constraint $Eq\ a$ can entail constraint $Eq\ [a]$ but not the other way around. In a system equipped with bidirectional instances, *cmp* can have multiple most

[7] An empty set of constraints can be logically interpreted as the constant *True*.

general types where none is more general than the other. All the following types are equally general:

$$\begin{array}{l} cmp :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool \\ cmp :: Eq\ [[a]] \Rightarrow a \rightarrow a \rightarrow Bool \\ cmp :: Eq\ [Maybe\ [a]] \Rightarrow a \rightarrow a \rightarrow Bool \end{array}$$

In fact, *cmp* has *infinitely many* principal types. This is not new. The Hindley-Damas-Milner system (Section 5.3.2) suffers from the same problem, as well as its extension with qualified types (Chapter 6).

For HM, principality of types is refined to take into account the possibilities for positioning universal quantifiers. For example, function f can have either of the following types

$$\begin{array}{ll} f :: \forall a.\ \forall b.\ a \rightarrow b \rightarrow a & \text{(signature 1)} \\ f :: \forall b.\ \forall a.\ a \rightarrow b \rightarrow a & \text{(signature 2)} \\ f = \lambda x.\ \lambda y.\ x & \end{array}$$

yet none is more general than the other.[8] Nevertheless, from one signature we can derive the other via substitution $[a/b, b/a]$.

Similarly, the basic system exhibits the same problem in terms of the order of constraints, as well as by means of *simplification* ((*Eq a*, *Ord a*) is isomorphic to *Ord a*).

In summary, in the presence of bidirectional instances a function can have infinitely many (isomorphic to each other) principal types. This is not necessarily a problem but in order to ensure well-defined semantics for our calculus, it is imperative that we refine the notion of *type subsumption* (see relation $\Gamma \models \sigma_1 \preceq \sigma_2$ in Section 5.3.3), as well as the definition of the *principal type* property (see Theorem 8).

9.2.5 Our Approach

In the remainder of this chapter we present the development of bidirectional instances and illustrate how we address the aforementioned challenges.

The key idea of our technique is to capture the parameter-dependent instance context in an open type-level function and store it within class dictionaries. To that end, our target language is not plain System F, but System F_C, which allows for the definition of open type functions. We elaborate on our approach in the next section.

[8]Notice though that the type-directed elaboration to System F we presented in Section 5.3.2 would elaborate f differently, depending on its type.

9.3 Bidirectional Instances

In this section we present the formalization of type classes with bidirectional instances. Since the system is a conservative extension of the basic system (Chapter 6), we only present the differences.

9.3.1 Syntax Extensions

Instead of directly introducing a logical biconditional connective into our calculus, we take a simpler approach: a class instance gives rise to both implications (left-to-right and right-to-left) separately. This allows us to reuse existing infrastructure and the well-established dictionary-passing elaboration method (Hall et al., 1996). For example, the equality instance for tuples

$$\textbf{instance}\ (Eq\ c, Eq\ d) \Rightarrow Eq\ (c, d)\ \textbf{where}\ \{\ (\texttt{==}) = \ldots\ \}$$

is interpreted as the following three axioms:[9]

$$\forall c.\ \forall d.\ (Eq\ c, Eq\ d) \Rightarrow Eq\ (c, d) \qquad \texttt{(IA)}$$
$$\forall c.\ \forall d.\ Eq\ (c, d) \Rightarrow Eq\ c \qquad \texttt{(BA1)}$$
$$\forall c.\ \forall d.\ Eq\ (c, d) \Rightarrow Eq\ d \qquad \texttt{(BA2)}$$

The first we call the *instance axiom*, and the other two the *inverted instance axioms*. In order to use the inverted axioms selectively and avoid the termination issues we mentioned earlier, we extend the syntax of program theory P with an additional component, the inverted instance axioms $\mathcal{A}_B$ (where B stands for "bidirectional"):

$$P ::= \langle \mathcal{A}_B, \mathcal{A}_S, \mathcal{A}_I, \mathcal{C}_L \rangle \qquad \textit{program theory}$$

As we illustrate below, similarly to superclass axioms $\mathcal{A}_S$, inverted instance axioms $\mathcal{A}_B$ are used only during type checking but not during type inference. The rest of the syntax is identical to the syntax of the basic system we presented in Figure 6.1.

[9]Instead of introducing a conjunction symbol into the logic, we generate two separate axioms using the distributivity of implication over conjunction:

$$(\phi_1 \rightarrow \phi_2 \wedge \phi_3) \leftrightarrow ((\phi_1 \rightarrow \phi_2) \wedge (\phi_1 \rightarrow \phi_3))$$

9.3.2 Specification Extensions

Similarly, the specification of typing and elaboration is for the most part identical to that of the basic system (Section 6.3). Since System F is a strict subset of System F_C, all relations presented in Section 6.3 remain unaltered.

The changes bidirectional instances introduce are concentrated in class and instance declaration typing, which we now discuss.

Class Declarations The specification of class typing with elaboration is given by judgment $\Gamma \vdash_{\text{CLS}} cls : P; \Gamma' \rightsquigarrow \overline{decl}$, as is for the basic system (Figure 6.2). Since for the most part the rule is identical to the basic system, we only highlight the differences. For a class declaration of the form

$$\textbf{class}\ \forall a.\ (Q_1, \ldots, Q_n) \Rightarrow \texttt{TC}\ a\ \textbf{where}\ \{\ f :: \sigma\ \}$$

we have the following:

1. Firstly, in addition to the superclass and method projections, the class declaration gives rise to a System F_C type family declaration:

   $$\textbf{type}\ F_{\texttt{TC}}\ a$$

   Function $F_{\texttt{TC}}\ a$ captures the functional dependency between the instance context and the class parameter. Hence, function $F_{\texttt{TC}}$ is populated by clauses whenever a `TC` instance is encountered.

2. Secondly, we extend the dictionary declaration, so that it can store the instance context of type $F_{\texttt{TC}}\ a$:

   $$\textbf{data}\ T_{\texttt{TC}}\ a = K_{\texttt{TC}}\ \boxed{(F_{\texttt{TC}}\ a)}\ \overline{\upsilon}^n\ \upsilon$$

   where $\overline{\upsilon}^n$ are the dictionary types corresponding to the superclass constraints $(Q_1, \ldots, Q_n)$ and υ is the elaboration of method type σ.

3. Finally, since the data constructor $K_{\texttt{TC}}$ now stores an additional field, we "shift" the superclass and method projections accordingly:

   $$\begin{array}{ll}\textbf{let}\ d_i : \forall a.\ T_{\texttt{TC}}\ a \to \upsilon_i = \Lambda a.\ \lambda(d : T_{\texttt{TC}}\ a).\ proj_{\texttt{TC}}^{i+1}(d) & i \in [1 \ldots n] \\ \textbf{let}\ f : \forall a.\ T_{\texttt{TC}}\ a \to \upsilon = \Lambda a.\ \lambda(d : T_{\texttt{TC}}\ a).\ proj_{\texttt{TC}}^{n+2}(d)] & \end{array}$$

Instance Declarations Typing for instance declarations also preserves the signature we gave in Figure 6.2. For a class instance of the form

$$\textbf{instance}\ \forall \overline{b}.\ (Q_1, \ldots, Q_m) \Rightarrow \texttt{TC}\ \tau\ \textbf{where}\ \{\ f = e\ \}$$

bidirectional instances introduce the following extensions:

1. Firstly, an additional clause is generated for function $F_{\mathtt{TC}}$, capturing the dependency between the instance parameter τ and instance context $(Q_1, \ldots, Q_m)$:
$$\textbf{axiom } g_\tau^{\mathtt{TC}}\ \overline{b} : F_{\mathtt{TC}}\ \upsilon \sim (\upsilon_1, \ldots, \upsilon_m)$$
where υ_i is the dictionary type representation of the i-th constraint in the instance context and υ is the elaboration of the type parameter τ.

2. Secondly, the program theory extension introduced by the instance now includes the inverted instance axioms, which take the form:
$$S_i = \forall \overline{b}.\ \mathtt{TC}\ \tau \Rightarrow Q_i \qquad i \in [1 \ldots m]$$
Of course, such implications need to be reflected in term-level functions in the generated System $\mathrm{F_C}$ code.

3. For every implication S_i, we generate a projection function d_i, given by the following definition:
$$\textbf{let } d_i : \forall \overline{b}.\ T_{\mathtt{TC}}\ \upsilon \rightarrow \upsilon_i = \Lambda \overline{b}.\ \lambda(d : T_{\mathtt{TC}}\ \upsilon).\ \mathit{ctxProj}^i_{\mathtt{TC}}(d) \qquad i \in [1 \ldots m]$$
where function $\mathit{ctxProj}^i_{\mathtt{TC}}(d)$ is defined as follows:
$$\begin{aligned}\mathit{ctxProj}^i_{\mathtt{TC}}(d) \equiv\ & \textbf{case } d \textbf{ of} \\ & \quad K_{\mathtt{TC}}\ \mathit{ctx}\ \overline{d}^n\ x \rightarrow \textbf{case } \mathit{ctx} \triangleright (g_\tau^{\mathtt{TC}}\ \overline{b}) \textbf{ of} \\ & \qquad\qquad (d_1, \ldots, d_m) \rightarrow d_i\end{aligned}$$
The outer pattern matching exposes the instance context ctx, of type $F_{\mathtt{TC}}\ \upsilon$, which we explicitly cast to a tuple of all instance context dictionaries: $(d_1, \ldots, d_m)$. Then, the inner pattern matching extracts and returns the corresponding instance context dictionary d_i.

4. Finally, the implementation of the instance dictionary (transformer) needs to store the instance context dictionaries within the dictionary for $\mathtt{TC}\ \tau$. Thus, the instance dictionary (transformer) now takes the form:
$$\textbf{let } d : \forall \overline{b}.\ \overline{\upsilon_i}^m \rightarrow T_{\mathtt{TC}}\ \upsilon = \Lambda \overline{b}.\ \lambda(\overline{d : \upsilon}^m).\ K_{\mathtt{TC}}\ \upsilon\ ((d_1, \ldots, d_m) \triangleright \gamma)\ \overline{t}^n\ t]$$
where $\gamma = \mathrm{sym}\ (g_\tau^{\mathtt{TC}}\ \overline{b})$. For the constructed dictionary to be well-typed, the tuple $(d_1, \ldots, d_m)$ containing all instance context dictionaries needs to be explicitly cast to have type $F_{\mathtt{TC}}\ \upsilon$, as the type of $K_{\mathtt{TC}}$ requires. This is exactly what coercion γ proves:
$$\gamma : (\upsilon_1, \ldots, \upsilon_m) \sim F_{\mathtt{TC}}\ \upsilon$$

9.3.3 Algorithm Extensions

Type inference is again for the most part identical to that of the basic system (Section 6.4). The changes that bidirectional instances introduce are concentrated in declarations. Type inference for class declarations is identical to its specification so we only discuss the differences in class instances and value bindings.

Instance Declarations Type inference for instance declarations behaves similarly to its specification: steps 1–3 above remain identical. The main difference lies in the type inference and subsumption checking for the method implementation.

In addition to the superclass closure ($\textsc{ScClosure}(\overline{a}, P) = (\mathcal{A}, \mathbb{E})$), we also compute the transitive closure of the inverted axioms. Thus, we replace function $\textsc{ScClosure}$ of Section 6.4 with function $\textsc{InvScClosure}$:

$$\begin{array}{l} \textsc{InvScClosure}(\overline{a}, \langle \mathcal{A}_B, \mathcal{A}_S, \mathcal{A}_I, \mathcal{C}_L \rangle) = ((\mathcal{C}'_L, \mathcal{A}_I, \mathcal{C}_L), \mathbb{E}) \\ \quad \text{where } (\mathcal{C}'_L, \mathbb{E}) = \mathit{mponens}^*(\overline{a}, (\mathcal{A}_B, \mathcal{A}_S), \mathcal{C}_L) \end{array}$$

Value Bindings Type inference for value bindings is also mildly affected by bidirectional instances. Top-level bindings without a type annotation simply ignore the inverted axioms, alongside the superclass axioms. That is, users that enable bidirectional instances can expect type inference to behave just as it did without them.

A top-level value binding with an explicit type annotation behaves differently, just as method typing does. Just as we do with superclass constraints, we also compute the transitive closure of the inverted axioms, making more derivations possible. For example, function *cmp* we presented earlier

$$\begin{array}{l} \mathit{cmp} :: \mathit{Eq}\ [a] \Rightarrow a \rightarrow a \rightarrow \mathit{Bool} \\ \mathit{cmp}\ x\ y = x \ \texttt{==}\ y \end{array}$$

is now well-typed and is elaborated as follows:

$$\begin{array}{l} \mathit{cmp} : \forall a.\ T_{\mathit{Eq}}\ [a] \rightarrow a \rightarrow a \rightarrow \mathit{Bool} \\ \mathit{cmp} = \Lambda a.\ \lambda(\mathit{dla} : T_{\mathit{Eq}}\ [a]).\ \lambda(x : a).\ \lambda(y : a). \\ \qquad\qquad \textbf{let}\ \mathit{da} : T_{\mathit{Eq}}\ a = \mathit{dinv}\ a\ d\ \textbf{in}\ (\texttt{==})\ a\ \mathit{da}\ x\ y \end{array}$$

where *dinv* is the inverted axiom of the *Eq* instance for lists:

$$\begin{array}{l} \mathit{dinv} : \forall a.\ T_{\mathit{Eq}}\ [a] \rightarrow T_{\mathit{Eq}}\ a \\ \mathit{dinv} = \Lambda a.\ \lambda(d : T_{\mathit{Eq}}\ [a]). \\ \qquad\qquad \textbf{case}\ d\ \textbf{of}\ \{\ K_{\mathit{Eq}}\ \mathit{ctx}\ \overline{d}\ x \rightarrow \textbf{case}\ \mathit{ctx} \triangleright (g^{\mathit{Eq}}_{[a]}\ a)\ \textbf{of}\ \{\ d' \rightarrow d'\ \}\ \} \end{array}$$

and $g^{Eq}_{[a]}\ a : F_{Eq}\ [a] \sim T_{Eq}\ a$.

Notice that in this case, the inner pattern match does not contribute anything, since the instance context contains a single constraint. In the general case though, the additional pattern matching is essential for separating the instance constraints. For example, for the *Eq* instance for tuples

$$\textbf{instance}\ (Eq\ c, Eq\ d) \Rightarrow Eq\ (c,d)\ \textbf{where}\ \{\ (==) = \ldots\ \}$$

the inverted instance axioms are elaborated into the following two functions

$$\begin{array}{l}
d_1 : \forall b.\ \forall c.\ T_{Eq}\ (b,c) \to T_{Eq}\ b \\
d_1 = \Lambda b.\ \Lambda c.\ \lambda(d : T_{Eq}\ (b,c)). \\
\qquad \textbf{case}\ d\ \textbf{of}\ \{\ K_{Eq}\ ctx\ x \to \textbf{case}\ (ctx \triangleright g_3\ b\ c)\ \textbf{of}\ \{\ (d'_1, d'_2) \to d'_1\ \}\ \}
\end{array}$$

$$\begin{array}{l}
d_2 : \forall b.\ \forall c.\ T_{Eq}\ (b,c) \to T_{Eq}\ c \\
d_2 = \Lambda b.\ \Lambda c.\ \lambda(d : T_{Eq}\ (b,c)). \\
\qquad \textbf{case}\ d\ \textbf{of}\ \{\ K_{Eq}\ ctx\ x \to \textbf{case}\ (ctx \triangleright g_3\ b\ c)\ \textbf{of}\ \{\ (d'_1, d'_2) \to d'_2\ \}\ \}
\end{array}$$

where the inner pattern match is essential for separating the instance context dictionaries d'_1 and d'_2.

9.4 Meta-theory

Finally, we now turn to the meta-theoretical properties of type classes with bidirectional instances. Since all judgments preserve the shape of the corresponding basic system judgments, the corresponding top-level theorem statements remain identical too.

Hence, we discuss below the two most interesting properties of the system: termination of type inference and the principal type property.

Termination To illustrate why type inference in the presence of bidirectional instances terminates, we first distinguish between type inference and type checking.

In cases where a type is inferred, the algorithm is identical to that of the basic system; the feature manifests itself when there are (implicit or explicit) type signatures. Hence, decreasing instance contexts are sufficient to ensure termination for cases where we only infer a type for an expression.

In cases where we need to check an expression against a type (e.g., in explicitly type-annotated terms or method implementations), the inverted axioms also come into play, as well as the superclass axioms.

Since we compute the closure of the superclass relation and the inverted axioms (by means of function InvScClosure), we need to ensure that both superclass and inverted axioms cannot be applied indefinitely.

For the former, the DAG restriction (from the termination conditions we presented in Section 6.5) is sufficient. For the latter, the decreasing contexts are also sufficient. To illustrate why, consider the following inverted axiom:

$$\forall a.\ \forall b.\ Eq\ (a, b) \Rightarrow Eq\ a$$

If we used the above axiom backwards (like simplification does), then the size of the type parameter would increase. Instead, during completion, we apply the axiom to constraints of the form $Eq\ (\tau_1, \tau_2)$, ending up with an additional axiom of a smaller size: $Eq\ \tau_1$. In short, usage of both the superclass and inverted axioms is bounded: the number of superclass axioms used is bounded by the height of the superclass DAG, and the number of inverted axioms by the size of the types in instance heads.

In fact, once the wanted constraints are fully simplified and the closure of the given constraints is computed, entailment can be simply implemented as set inclusion: all wanted constraints should be present in the set of givens.

To summarize, we are confident that type inference is terminating in the presence of bidirectional instances:

Conjecture 8 (Termination)**.** *If the source program satisfies the termination conditions of Section 6.5, then type inference for type classes with bidirectional instances terminates.*

Principality of Types As we discussed in Section 9.2.4, bidirectional instances require us to revisit the notion of principal types.

First, the specification of the principal type can remain identical to the corresponding for the basic system (see Section 6.4): the definition of a principal type does not specify one type, but rather the properties of it.

Second, type inference infers *one* principal type. Since plain type inference does not exploit the inverted axioms, the algorithm infers backwards-compatible principal types. Backwards-chaining simplifies constraints such as $Eq\ [a]$ to $Eq\ a$ but not the other way around. Thus, the algorithm would never infer a type of the form

$$\forall a.\ \forall b.\ Eq\ (a, b) \Rightarrow \ldots$$

but would infer the isomorphic (and also principal) type

$$\forall a.\ \forall b.\ (Eq\ a, Eq\ b) \Rightarrow \ldots$$

Expressions with explicit type annotations have only one principal type (the one specified by the signature). In these cases, the algorithm will use the inverted axioms to entail the wanted constraints $(Eq\ a, Eq\ b)$ using the given $Eq\ (a, b)$, thus constructing again the principal type.

A Note on Soundness An aspect of bidirectionality we have not discussed is its meta-theoretical properties in the presence of overlapping instances. It is known that overlapping instances make the semantics of type classes incoherent but they do not introduce unsoundness. In the presence of bidirectional instances, this is no longer true.

Take for instance the following overlapping instances:

$$\begin{array}{ll} \textbf{instance}\ Eq\ a \Rightarrow Eq\ [a] & (1) \\ \textbf{instance}\ Eq\ [a] & (2) \end{array}$$

Each instance gives rise to a type-equality axiom:

$$\begin{array}{l} \textbf{axiom}\ g_1\ a : F_{Eq}\ [a] \sim T_{Eq}\ a \\ \textbf{axiom}\ g_2\ a : F_{Eq}\ [a] \sim () \end{array}$$

Axioms g_1 and g_2 violate the compatibility condition (Definition 8), which in turn means that our elaboration would give rise to unsound System $\mathrm{F_C}$ code. Indeed, coercion $(\mathrm{sym}\ (g_1\ Int)) \mathbin{⨟} (g_2\ Int)$ is a proof of $T_{Eq}\ Int \sim ()$.

Of course, instances that we do not elaborate bidirectionally can overlap freely (resulting in sound, yet incoherent semantics). One could consider explicitly marking classes whose instances are to be interpreted bidirectionally and require only these to not overlap; we leave such design choices to compiler developers that consider implementing our feature.

9.5 Scientific Output

In this chapter we have presented a conservative extension of type classes, which allows for type classes to be interpreted bidirectionally. This extension significantly improves the interaction of GADTs with type classes, by allowing proper structural induction GADTs, even in the presence of qualified types.

The type inference and elaboration algorithm we presented in this chapter is also implemented in a prototype compiler, available at

`https://github.com/gkaracha/bidirectional-impl`

The prototype incorporates higher-kinded datatypes and performs type inference, elaboration into System F_C, and type checking of the generated code.

The contents of this chapter constitute part of ongoing work, which has been performed by the author of this work, in collaboration with Koen Pauwels, Michiel Derhaeg, and Tom Schrijvers. The prototype is based on the initial implementation of quantified class constraints (by the author of this thesis and Gert-Jan Bottu). The simplification of the prototype (to plain type classes without quantified class constraints), the extension of the target language to System F_C, and the extension with bidirectional instances has been implemented by Michiel Derhaeg, under the supervision of the author of this thesis.

Chapter 10

Conclusion

"So long, and thanks for all the fish."

—Douglas Adams, The Hitchhiker's Guide to the Galaxy, Vol. I

In this chapter, we briefly revisit the goals of this thesis and summarize the results we presented in each chapter. Additionally, we discuss ongoing work as well as possible ideas for future work based on the findings we presented. Since Parts I and II are fairly independent, we discuss all the above for each part separately.

10.1 Aim of the Thesis

The aim of this thesis is to improve upon existing and develop new means for type-level computation and reasoning. More specifically, we focus on two existing features: *pattern matching* (Part I) and *type classes* (Part II).

The goal of Part I is to improve upon existing mechanisms for reasoning about pattern matching. Pattern matching has been extended with a multitude of features throughout the years, yet we know of no algorithm that can reason about pattern matching properties in the presence of such features. Thus, the goal of Part I is the development of a pattern match checking algorithm that takes into account GADTs, guards, and laziness.

Part II focuses on the development of (mostly) new features that allow for more expressive type-level computation. More specifically, the overall goal of Part II is to lift the expressive power of type classes from Horn clauses to a significant fragment of first-order logic.

10.2 Summary

In this section we give an overview of the most significant results presented in this thesis. The results of Part I and Part II are presented Sections 10.2.1 and 10.2.2, respectively.

10.2.1 Pattern Matching

Part I focuses on the semantics of pattern matching and its meta-theoretical properties. More specifically, it addresses an old problem in a new setting: detection of missing and redundant clauses in the context of lazy pattern matching, extended with guards and GADTs.

Chapter 2: Research Question

> *"In mathematics the art of proposing a question must be held of higher value than solving it."*
>
> *—Georg Cantor*

First, Chapter 2 consists mostly on basic background information related to pattern matching and the extensions it has seen in the last years. Though most of the material in Chapter 2 is either folklore or part of published work, the main contribution of the chapter is the identification of the challenges one needs to address in order to reason about modern pattern matching in a lazy context.

Chapter 3: Pattern Match Checking Algorithm

Second, Chapter 3 addresses the problem outlined in Chapter 2 by developing a pattern match checking algorithm.

At the core of the algorithm lies a simple yet powerful pattern language, in which we can encode existing pattern matching extensions supported by GHC. The pattern language is an interesting research result on its own, and—as we discuss in more detail below—provides interesting opportunities for more extensions, such as non-linear patterns and *or-patterns*.

Next, Chapter 3 provides an algorithm that provides warnings for functions with redundant or missing patterns. These warnings are accurate, even in the presence of GADTs, guards, and laziness. The algorithm is concise, easy to understand, and—most importantly—modular. More specifically, the algorithm handles the structural aspects of pattern matching generically and relies on external satisfiability oracles for the non-structural aspects.

This separation of concerns is extremely helpful: one can adjust the expressivity/performance ratio of the algorithm simply by using different external oracles, without affecting the main implementation of the algorithm. Indeed, as we illustrate in Section 3.4, we can state the correctness of the algorithm by treating the oracles as a black box; as long as they are conservative,[1] the algorithm is sound with respect to Haskell's semantics.

Chapter 4: GHC Implementation

"There's a difference between knowing the path and walking the path."

—Morpheus, The Matrix

Finally, Chapter 4 shows how to take the algorithm of Chapter 3 from a theoretical model to an actual implementation in an industrial-strength compiler as GHC.

Many theoretical developments in the literature sustain heavy adaptations in order to be implemented in projects of this scale. The goal of Chapter 4 is to illustrate how the algorithm of Chapter 3 needs only minimal changes to be adopted by a general-purpose functional language compiler.

Additionally, Chapter 4 provides qualitative and quantitative performance results and insights we have obtained throughout the development of the algorithm in GHC. These results are of importance for the academic community at large: they motivate the design choices we made in the development of the algorithm and provide insight into performance-related aspects of pattern match checking in general.

Lastly, this artefact is available to all Haskell users; the implementation of our algorithm is part of the GHC distribution.

[1] By conservative we mean that they do not give false negative errors: if the oracle marks a set of constraints as unsatisfiable, then it should be indeed unsatisfiable.

10.2.2 Type Classes

Part II focuses on the semantics of type classes and develops three type class extensions, with an aim to (a) lift the expressive power of type classes to a significant fragment of first-order logic, and (b) offer new means for type-level computation to Haskell users.

Chapter 5: Background

Chapter 5 consists on background information related to the notions of type inference and qualified types, in the context of type classes. The material presented in Chapter 5 consists entirely of folklore and published work and summarizes the expected semantics of type classes as well as relevant notions.

Chapter 6: The Basic System

Chapter 6 presents a technical (yet easy-to-understand) formalization of type classes which we use to develop our extensions on in the subsequent chapters. The main contribution of this chapter lies in the handling of superclasses, an important aspect of type classes that is often omitted; to our knowledge, we are the first to formalize type inference and elaboration of type classes in the presence of superclass constraints.

Moreover, prior work on type classes makes simplifications for the sake of brevity and readability; the system we present in Chapter 6 takes no such shortcuts. We believe that this system is a great candidate for proving meta-theoretical properties of type classes, in a way that also reflects their implementations. This constitutes part of our future work, as we discuss below.

Chapter 7: Quantified Class Constraints

The first extension of type classes (*Quantified Class Constraints*) is given in Chapter 7. This extension allows universal quantification and logical implications to appear in arbitrarily nested positions within formulas, thus lifting the expressing power of type classes from simple Horn clauses to the universal fragment of Hereditary Harrop logic.

The main contribution of this chapter lies in the typing specification of the feature, which utilizes a technique known as “focusing”, to provide a simple and expressive semantics. Next, we provide a sound and complete algorithm, which also elaborates source programs in a language akin to System F. Last, we give a detailed account of how to ensure termination of type inference in the presence of quantified class constraints and provide a prototype implementation of the type inference algorithm.

Due to the popularity of and the numerous requests for the development of this feature, we expect an implementation of this feature in GHC to have a big impact on Haskell users. Hence, such an implementation constitutes part of our future work.

Chapter 8: Functional Dependencies

The second extension of type classes (*Functional Dependencies*) is given in Chapter 8. This extension allows the decoration of multi-parameter type classes with explicit annotations of parameters uniquely determining other parameters.

This feature can be used to resolve ambiguities in type inference and as a means of type-level computation (a functional dependency can be interpreted as an open type-level function).

Functional dependencies have been introduced in Haskell in the early 2000s, yet several aspects of their semantics have been unspecified. The main novelty of our work lies in the interpretation of functional dependencies as explicitly-named type-level functions. This allows us to give the first algorithm for elaborating functional dependencies into a statically-typed language akin to System F (System F_C). Furthermore, our type inference and elaboration algorithm uses the same building blocks as the one for type families, thus making possible the coexistence of (and interaction between) both features within the same system.

Chapter 9: Bidirectional Instances

The third and last extension of type classes (*Bidirectional Instances*) is given in Chapter 9. This extension allows class instances to be interpreted and elaborated bidirectionally, effectively extending the logical interpretation of type classes with the biconditional connective.

The main novelties of this work are (a) the encapsulation of the functional dependency between the instance parameters and the instance context within an open type-level function, and (b) the extension of the traditional dictionary-passing translation of type classes with an additional field, which captures the parameter-dependent set of instance context dictionaries.

This extension significantly raises the expressive power of type classes, particularly from the programmer's-eye view. More specifically, the bidirectional interpretation of type class instances allows functions with qualified types to perform structural induction over GADTs. Additionally, addressing this infelicity raises the expressive power of all type-class-based features, such as functional dependencies and associated type synonyms.

10.3 Ongoing and Future Work

"One never notices what has been done; one can only see what remains to be done."

—Marie Curie

We see a lot of potential for further development of both parts of this thesis. Indeed, several of the ideas we present in this section are already under active development.

10.3.1 Pattern Matching

Pattern Match Compilation and Optimizations

We would like to work mainly on two approaches to efficient pattern match compilation.

First, we would like to integrate pattern match checking and compilation so that multiple traversals can be avoided. Currently, they are performed separately (see Chapter 4), which (a) requires two traversals, and (b) makes it difficult to transfer exhaustiveness and redundancy information obtained by the checker to the pattern match compiler.

Unfortunately, the algorithm of Chapter 3 traverses the *pattern matrix* in a row-by-row fashion (according to the semantics of pattern matching), while existing algorithms for efficient pattern match compilation (see for example the work of Maranget and Para (1994), and Wadler (1987a)) perform more of a column-based traversal. This makes the integration of the two within the same algorithm a challenging task.

Alternatively, we would like to investigate ways of exploiting the analysis results from pattern match checking for optimized compilation, without combining the two algorithms. Currently, the two are entirely disconnected but it is well-established that compilation can significantly benefit from pattern match checking (see for example the work of Le Fessant and Maranget (2001), which is actively used by the OCaml compiler).

Source Language Extensions

As illustrated throughout the thesis, our core pattern language is concise and expressive enough to encode a multitude of pattern matching features. More specifically, the ability of the pattern language to encode view patterns and the algorithm to reason about them hint at possible source-level pattern matching extensions (based on view patterns) we could implement.

Several encodings of extensions using view patterns are already known within the GHC community[2]. Their encoding using view patterns points at ways to compile them into primitive pattern matching, as well as how to translate them into our pattern language.

For example, we can translate *"and-patterns"* of the form ($\mathbb{P}_1$ & $\mathbb{P}_2$) (also known as *both-patterns*) straightforwardly, by adding the following clause to function $\mathit{translate}_p(\cdot)$ of Section 3.2.2:

$$\mathit{translate}_p(\mathbb{P}_1 \;\&\; \mathbb{P}_2) = x\ (\mathit{translate}_p(\mathbb{P}_1) \leftarrow x)\ (\mathit{translate}_p(\mathbb{P}_2) \leftarrow x)$$

(an expression matches a pattern of the form ($\mathbb{P}_1$ & $\mathbb{P}_2$) if it matches both $\mathbb{P}_1$ and $\mathbb{P}_2$).

Furthermore, we could also consider the extension of Haskell with non-linear patterns (the semantics being that the first appearance of a variable *binds* and the subsequent ones *check*). For example, function f

$$\begin{array}{lcl} f\ x\ x & = & \mathit{True} \\ f\ y\ z & = & \mathit{False} \end{array}$$

would be assigned the type

$$f :: \mathit{Eq}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Bool}$$

and the clauses would be translated into

$$\begin{array}{lcl} x\ x'\ (\mathit{True} \leftarrow x == x') & \rightarrow & \mathit{True} \\ y\ z & \rightarrow & \mathit{False} \end{array}$$

This semantics is just one of many possible ones; the key idea here is the numerous possibilities that our pattern language and checking algorithm provide.

[2]See for example `https://ghc.haskell.org/trac/ghc/wiki/ViewPatterns#Examples`.

Pattern Trees and Or-patterns

As we illustrated in Section 4.1.2, our implementation in GHC generalizes pattern vectors to a tree-like structure, to account for multiple guard-vectors associated with the same pattern vector. This generalization provides interesting opportunities for follow-on work, such as a full-blown generalization of pattern matrices to *pattern trees*.

This natural generalization allows the encoding of and reasoning about *or-patterns*, a feature already present in several functional languages (e.g., OCaml (Maranget, 2007) and Coq (The Coq development team, 2004)), but not Haskell.

Indeed, we have already extended the pattern language of Section 3.2 and the checker of Section 3.3 to accommodate pattern trees and are planning on publishing our results in the near future.

Bug Fixes

Though we have successfully integrated our implementation of the pattern match checking algorithm in GHC, there are still a couple of infelicities that would like to address in the future. More specifically, we would like to address the following GHC ticket (bug report):

Ticket #12949 Both the pattern translation and the algorithm of Chapter 3 assume that the algorithm runs *after* type classes are translated into their dictionary counterparts.

Though this is indeed the case (the algorithm is deployed post-type-inference), the algorithm currently does not tap into this information: two calls of an overloaded function with different type instantiations represent different terms and the term oracle needs to treat them as such. This is an implementation bug we would like to address in the near future.

Performance Improvements

Another line of work we would like to pursue in the future is concerned with improving the performance and space consumption of our implementation.

Firstly, we would like to investigate the benefits of solving type-level constraints incrementally, just as we do with term-level constraints. Incremental solving for term constraints has shown a significant performance improvement—especially in the presence of literals and pattern guards (Section 4.3)—so we expect to get similar improvements with incremental solving of type constraints.

Secondly, we would like to improve the expressiveness of the term-oracle. As we have learned through the bug reports we discussed in Section 4.3, the performance of the oracles is critical for the overall performance of the algorithm. Thus, we would like to investigate ways for our term oracle to efficiently deal with constraints derived from view patterns.

Thirdly, we would like to investigate a different representation of value set abstractions, as prefix trees. Indeed, the traversal pattern of the algorithm naturally gives rise to value set abstractions with shared prefixes so we expect this representation to be more space-efficient. Though our initial implementation shared common prefixes using this alternative representation, the performance cost was severe, which led us to the less sophisticated representation of value set abstractions as lists of value vector abstractions. Our aspiration to return to this more elegant representation is captured in bug report #11528.

Pattern Synonyms

Recent work by Pickering et al. (2016) formalized the semantics of *pattern synonyms*, which differs from simple macro expansion. Given a pattern synonym declaration of the form

$$\mathbf{pattern}\ \mathbf{P}\ x_1\ \ldots\ x_n = \mathbb{P}$$

an expression matched against pattern $(\mathbf{P}\ \mathbb{P}_1 \ldots \mathbb{P}_n)$ first matches against pattern $\mathbb{P}$, and if successful, it proceeds with matching the sub-terms against sub-patterns $\mathbb{P}_1 \ldots \mathbb{P}_n$. Using this semantics, we can easily extend our desugaring algorithm of Figure 3.3 to handle pattern synonyms too:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{translate}_p(\mathbf{P}\ \mathbb{P}_1 \ldots \mathbb{P}_n) = z\ \vec{p_0}\ \vec{p_1}\ \ldots\ \vec{p_n}} \\ \text{where} & \mathbf{pattern}\ \mathbf{P}\ x_1\ \ldots\ x_n = \mathbb{P} \\ & \vec{p_0} = \mathit{translate}_g([\overline{y}/\overline{x}]\mathbb{P} \leftarrow z) \\ & \vec{p_i} = \mathit{translate}_g(\mathbb{P}_i \leftarrow y_i) \qquad (i \in [1 \ldots n]) \end{array}$$

Nevertheless, reasoning about pattern synonyms by exposing their definition defeats the main purpose of their design: abstraction and modularity.

One recent attempt to tackle coverage checking in the presence of pattern synonyms using user-specified pragma `COMPLETE`[3] is still under-specified and

[3] A `COMPLETE` pragma definition axiomatically declares a collection of data constructors and/or pattern synonyms to constitute a signature of a type constructor.

its implementation suffers from several problems due to the lack of a proper specification.[4]

Thus, in the near future we would like to formally specify how our coverage checking algorithm can deal with pattern synonyms in a principled way, without exposing their underlying definition.

10.3.2 Type Classes

Implementation of Functional Dependencies in GHC

> *"Walking on water and developing software from a specification are easy if both are frozen."*
>
> —*Edward V. Berard*

The results we presented in Chapter 8 enable the proper integration of functional dependencies in Haskell's eco-system of advanced type-level features. We would like to pursue this line of work in the near future and even address feature request #11534, where the reconstructed functional dependency witness is exposed to the user for explicit use.

To this end, we are already developing a standalone compiler prototype with functional dependencies (the main developer of this work is Michiel Derhaeg, under the author's supervision). In future work we plan on extending the prototype with type families and GADTs, in order to study their interaction at length and possibly revise the formalization of Chapter 8.

Once the development of the prototype comes to an end, we expect to have (a) qualitative and quantitative results on the interaction of the aforementioned features, and (b) a stable technical specification of the algorithm, which we can use to guide an implementation in GHC.

Bidirectional Instances

Currently we are also implementing bidirectional instances in our prototype compiler with functional dependencies. Since the extension is rather small but quite impactful, we expect to have a complete implementation of bidirectional instances and functional dependencies soon, which will allow us to study the

[4]See for example the open tickets in the pattern match checking-related webpage: `https://ghc.haskell.org/trac/ghc/wiki/PatternMatchCheck`.

interaction of the two features and identify more benefits of interpreting instances bidirectionally.

From Type Classes to First-order Logic

If one considers the syntax of first-order logic we presented in Section 5.4.2, the extensions of Chapters 7, 8, and 9 already take a significant step towards bridging the gap between type classes and first-order logic:

Chapter 7 significantly generalizes the positions in which universal quantification and implication can appear in a formula. Chapter 8 extends the type language with function symbols, as well as the language of formulas with type equalities and existential quantification. Finally, Chapter 9 introduces bidirectionality, effectively adding the biconditional symbol to the language of formulas.

Nevertheless, each feature has been developed in isolation. This allows us to study the delicacies of each feature but does not give us any indication of whether the cohabitation of all features in the same system would be possible.

We can already pinpoint the main challenge in this quest: *how to entail equality constraints in the presence of backtracking*. To our knowledge, no existing constraint solver can deal with both, and OutsideIn(X) (Vytiniotis et al., 2011), the solver we instantiate the entailment specification of Sections 8.5.4 and 8.5.5 with, certainly does not. Integration of bidirectional instances with functional dependencies seems straightforward: inverted axioms derived from bidirectionality can be exploited first (as happens with superclass constraints), and OutsideIn(X) can be utilized later to resolve wanted constraints. The source of the problem lies in the integration of quantified constraints: the completeness of the entailment algorithm for quantified constraints (Figure 7.5) relies on backtracking. What's more, as we discussed in Section 9.2.3, treating inverted axioms like other axioms makes constraint solving non-terminating. Thus, a backtracking-based approach cannot work in the presence of bidirectional instances.

In summary, the research question we need to address is the following: *is there a way to entail class and equality constraints in the presence of backtracking?* We would like to answer this research question in the near future.

Constraint Disjunction

Another line of work we plan for the future is the development and integration of another feature with those we have presented in this thesis (quantified constraints, functional dependencies, and bidirectional instances): *Constraint Disjunction.*

In short, the only[5] connective of first-order logic that is not supported by our extensions is logical disjunction. Indirectly, overlapping instances can introduce constraint disjunction (since there can be multiple ways to satisfy the same constraint, but in an incoherent way (see Section 6.5)).

In contrast to overlapping instances, by constraint disjunction we refer to the introduction of a biased constraint operator $\vec{\vee}$ with a coherent semantics. As an example, consider the following function which specializes its implementation based on the available constraints:

$$\begin{aligned} &nub :: (Ord\ a\ \vec{\vee}\ Eq\ a) \Rightarrow [a] \rightarrow [a] \\ &nub\ =\ map\ head\ .\ group\ .\ sort \\ &\qquad\ \vec{\vee}\ \ Data.List.nub \end{aligned}$$

The type-level operator $\vec{\vee}$ reflects the two implementations given for *num*, which are separated by a term-level operator $\vec{\vee}$.[6] The intended semantics for *nub* is that the first implementation is preferred if *Ord a* can be established. If only the weaker constraint *Eq a* is satisfiable, the compiler specializes a call to *nub* to the second implementation. Notice that the first implementation has a run-time complexity of $O(nlogn)$, while the second runs in $O(n^2)$.

The two most notable benefits of this approach are the following:

Overloading As is already achieved with type classes, constraint disjunction promotes function overloading, yet in a slightly different fashion: one can now use a single name for multiple implementations, even for the same type (modulo the qualification). For example, this would obviate the need for both functions *nub* (from package `Data.List`) and *nubOrd* (from package `Data.List.Extra`).

Separation of Concerns Using constraint disjunction, the choice of the appropriate implementation becomes a concern of the *call-site*, and not the definition. This allows the separate development of libraries, independently of their use: the compiler can choose the most performant implementation automatically, at call-sites.

[5]If one were to ignore negation ($\neg$).

[6]The use of the same symbol at the level of types and terms is incidental.

Meta-theory

Furthermore, for each of the features we introduced we have only partially proven their meta-theoretical properties. In the near future, we would like to formally prove (and possibly mechanize the proofs of) the meta-theoretical properties we have stated in every chapter.

The meta-theory of quantified class constraints is already being under development, mainly by Gert-Jan Bottu, Tom Schrijvers, and Ningning Xie. The meta-theory of bidirectional instances is also under development, mainly by Koen Pawels (under the supervision of the author of this thesis), in collaboration with Gert-Jan Bottu, Tom Schrijvers, and Ningning Xie.

Moreover, if the four extensions (quantified class constraints, functional dependencies, bidirectional instances, and constraint disjunction) can coexist within the same system, we would like to prove the corresponding meta-theoretical properties for the complete system.

From Type Classes to a Fragment of Second-order Logic

Last but not least, we would like to investigate the extension of our systems with quantification over predicates[7], effectively raising the power of type classes to (a fragment of) second-order logic. Alternatively, one might achieve similar results by combining the results we presented in Chapter 7, with GHC extension `ConstraintKinds`, but none of the two approaches is studied yet. We would like to pursue this line of work in the near future.

[7]See for example GHC feature request #5927.

Appendix A

Basic System: Additional Judgments

A.1 Program and Declaration Typing

The specification of program typing (with elaboration) for the basic system we omitted in Section 6.3 is presented in Figure A.1. Essentially, judgment $\vdash_{\texttt{PGM}} pgm : P; \Gamma \rightsquigarrow \overline{decl}$ checks a program pgm, via the auxiliary judgment $P; \Gamma \vdash_{\texttt{DCL}} \overline{decl} : P'; \Gamma' \rightsquigarrow \overline{decl}$. The latter, straightforwardly combines the judgments we gave in Section 6.3 ($\Gamma \vdash_{\texttt{CLS}} cls : P; \Gamma' \rightsquigarrow \overline{decl}$ for class declarations, $P; \Gamma \vdash_{\texttt{INS}} ins : P' \rightsquigarrow decl$ for instance declarations, and $P; \Gamma \vdash_{\texttt{VAL}} val : \Gamma' \rightsquigarrow decl$ for value bindings) to check each kind of declaration. In parallel, it combines the environment extensions induced by each declaration.

Figure A.1 Basic System: Program Typing with Elaboration

$\boxed{\vdash_{\texttt{PGM}} \mathit{pgm} : P; \Gamma \rightsquigarrow \overline{\mathit{decl}}}$ Program Typing

$$\frac{\bullet; \bullet \vdash_{\texttt{DCL}} \overline{\mathit{decl}} : P; \Gamma \rightsquigarrow \overline{\mathit{decl}}}{\vdash_{\texttt{PGM}} \overline{\mathit{decl}} : P; \Gamma \rightsquigarrow \overline{\mathit{decl}}} \textsc{Pgm}$$

$\boxed{P; \Gamma \vdash_{\texttt{DCL}} \overline{\mathit{decl}} : P'; \Gamma' \rightsquigarrow \overline{\mathit{decl}}}$ Declaration Typing

$$\frac{}{P; \Gamma \vdash_{\texttt{DCL}} \bullet : P; \Gamma \rightsquigarrow \bullet} \textsc{NilD}$$

$$\frac{\Gamma \vdash_{\texttt{CLS}} \mathit{cls} : \Gamma_c; P_c \rightsquigarrow \mathit{decl}_c \qquad P, P_c; \Gamma, \Gamma_c \vdash_{\texttt{DCL}} \overline{\mathit{decl}} : P_d; \Gamma_d \rightsquigarrow \overline{\mathit{decl}_d}}{P; \Gamma \vdash_{\texttt{DCL}} (\mathit{cls}; \overline{\mathit{decl}}) : P_d; \Gamma_d \rightsquigarrow \overline{\mathit{decl}_c}; \overline{\mathit{decl}_d}} \textsc{ClsD}$$

$$\frac{P; \Gamma \vdash_{\texttt{INS}} \mathit{ins} : P_i \rightsquigarrow \mathit{decl}_i \qquad P, P_i; \Gamma \vdash_{\texttt{DCL}} \overline{\mathit{decl}} : P_d; \Gamma_d \rightsquigarrow \overline{\mathit{decl}_d}}{P; \Gamma \vdash_{\texttt{DCL}} (\mathit{ins}; \overline{\mathit{decl}}) : P_d; \Gamma_d \rightsquigarrow \mathit{decl}_i; \overline{\mathit{decl}_d}} \textsc{InsD}$$

$$\frac{P; \Gamma \vdash_{\texttt{VAL}} \mathit{val} : \Gamma_v \rightsquigarrow \mathit{decl}_v \qquad P; \Gamma, \Gamma_v \vdash_{\texttt{DCL}} \overline{\mathit{decl}} : P_d; \Gamma_d \rightsquigarrow \overline{\mathit{decl}_d}}{P; \Gamma \vdash_{\texttt{DCL}} (\mathit{val}; \overline{\mathit{decl}}) : P_d; \Gamma_d \rightsquigarrow \mathit{decl}_v; \overline{\mathit{decl}_d}} \textsc{ValD}$$

Appendix B

Functional Dependencies: Additional Material

This chapter presents additional material related to Chapter 8.

B.1 Constraint Schemes, CHRs and System $\mathsf{F_C}$

In this section we informally illustrate that both our specification of typing for functional dependencies (Section 8.3) and our elaboration algorithm (Section 8.5) have semantics compatible with the Constraint Handling Rules of Sulzmann et al. (2007b).

B.1.1 Class CHRs

According to Sulzmann et al., a class declaration

$$\textbf{class } \forall \overline{a}\overline{b}.\ \overline{\pi} \Rightarrow \texttt{TC}\ \overline{a} \mid fd_1, \dots, fd_m$$

gives rise to two kinds of constraint handling rules:

Class CHR. The class chr takes the form **rule** $\texttt{TC}\ \overline{a} \implies \overline{\pi}$, that is, almost the same as Scheme CS1a:

$$S_{C_\pi} = \forall \overline{a}.\ \texttt{TC}\ \overline{a} \Rightarrow \theta(\pi) \quad \forall \pi$$

where $\theta = det(\overline{a}, \overline{\pi})$. The main difference between the two is the substitution θ, which essentially replaces all skolem variables $\overline{b}$ in $\overline{\pi}$ with their (known) counterparts.[1]

The CHR has a direct interpretation into System $\mathrm{F_C}$, as a match context:

$$\mathbb{E} = \mathbf{case}\ (d : T_{\mathtt{TC}}\ \overline{a})\ \mathbf{of}\ \{\ \dots\ \}$$

Within the scope of $\mathbb{E}$, both $\overline{b}$ and $\overline{\pi}$ are available. The elaboration of the constraint scheme is slightly more complex. Matching against all superclass constraints $\overline{\pi}$ recursively makes available all coercions that connect $\overline{b}$ with $\overline{a}$. Composed, they can be used to cast the type of the superclass constraints to $\theta(\pi)$. As an example, consider the following definitions:

$$\begin{array}{l} \mathbf{class}\ C\ a\ b \mid a \to b \\ \mathbf{class}\ C\ a\ b \Rightarrow D\ a \end{array}$$

The corresponding constraint scheme is

$$\forall a.\ D\ a \Rightarrow C\ a\ (F_C\ a)$$

and is witnessed by the following function:

$$\begin{array}{l} f = \Lambda a.\ \lambda(d_1 : T_D\ a).\ \mathbf{case}\ d_1\ \mathbf{of} \\ \qquad T_D\ b\ (d_2 : T_C\ a\ b) \to \mathbf{case}\ d_2\ \mathbf{of} \\ \qquad\qquad T_C\ (\omega : F_C\ a \sim b) \to d_2 \triangleright \langle T_C \rangle\ \langle a \rangle\ (\mathrm{sym}\ \omega) \end{array}$$

This explains why we strictly require that $\overline{b} \subseteq dom(\theta)$: if the restriction does not hold, there are superclass constraint schemes that cannot be elaborated in System $\mathrm{F_C}$. Such declarations are ambiguous (and thus rejected anyway), so this restriction does not affect the expressive power of the system.

Functional Dependency CHRs. For each functional dependency $fd_i \equiv a_{i_1} \dots a_{i_n} \to a_{i_0}$, we have **rule** $\mathtt{TC}\ \overline{a}, \mathtt{TC}\ \theta(\overline{b}) \implies a_{i_0} \sim b_{i_0}$, where

$$\theta(b_j) = \begin{cases} a_j & \text{, if } j \in \{i_1, \dots, i_n\} \\ b_j & \text{, otherwise} \end{cases}$$

This rule is derivable using Scheme CS1b twice as follows:

$$\dfrac{\dfrac{\dfrac{\mathtt{TC}\ \overline{a}}{F_{\mathtt{TC}_i}\ \overline{a}^{i_n} \sim a_{i_0}}\ \textsc{1b}}{a_{i_0} \sim F_{\mathtt{TC}_i}\ \overline{a}^{i_n}}\ \textsc{Sym} \qquad \dfrac{\dfrac{\mathtt{TC}\ \theta(\overline{b})}{F_{\mathtt{TC}_i}\ \theta(\overline{b}^{i_n}) \sim \theta(b_{i_0})}\ \textsc{1b}}{F_{\mathtt{TC}_i}\ \overline{a}^{i_n} \sim b_{i_0}}\ \theta}{a_{i_0} \sim b_{i_0}}\ {}^{\circ}_{9}$$

[1]If we restrict ourselves to cases where $\overline{b} \subseteq dom(\theta)$, both our specification and the inference (with elaboration) are well-behaved.

The System F$_\mathrm{C}$ counterpart of this constraint scheme is the combination of two match contexts

$$\begin{aligned}\mathbb{E} = \mathbf{case}\ &(d_1 : T_{\texttt{TC}}\ \overline{a})\ \mathbf{of}\\ &T_{\texttt{TC}}\ \ldots\ \overline{\omega}_1\ \cdots \to \mathbf{case}\ (d_2 : T_{\texttt{TC}}\ \theta(\overline{b}))\ \mathbf{of}\\ &\qquad T_{\texttt{TC}}\ \ldots\ \overline{\omega}_2\ \cdots \to \Box\end{aligned}$$

and a *local* coercion $\gamma = (\text{sym}\ \omega_{i1})\ ⨾\ \omega_{i2}$. Notice the importance of the match context, for the well-scopedness of the coercion: sub-coercions ω_{i1} and ω_{i2} are available only within the scope of the match context.

B.1.2 Instance CHRs

Let there be an instance declaration

$$\mathbf{instance}\ \forall \overline{a}\overline{b}.\ \overline{\pi} \Rightarrow \texttt{TC}\ \overline{u}$$

According to Sulzmann et al., it also gives rise to two kinds of constraint handling rules:

Instance CHR. The instance rule takes the form

$$\mathbf{rule}\ \texttt{TC}\ \overline{u}\ \iff\ \overline{\pi}$$

which, according to CHR semantics, means that instead of proving $\texttt{TC}\ \overline{u}$, one suffices to prove $\overline{\pi}$. That is, we can always derive $\texttt{TC}\ \overline{u}$ from $\overline{\pi}$. This directly corresponds to Scheme CS2a:

$$S_{I_{\overline{\pi}}} = \forall \overline{a}.\ \theta(\overline{\pi}) \Rightarrow \texttt{TC}\ \overline{u}$$

where $\theta = \mathit{det}(\overline{a}, \overline{\pi})$. The interpretation of this scheme is the expected dictionary constructor, where $\overline{b}$ are appropriately instantiated. For a system that does not support the *type substitution property*, this wouldn't necessarily be accepted, since the quantification over $\overline{b}$ is non-parametric. Yet, as we illustrated in Chapter 8, our system does, and our instantiation is (by construction) the expected.

Instance Improvement CHRs. For each functional dependency $fd_i \equiv a_{i_1} \ldots a_{i_n} \to a_{i_0}$, we have $\mathbf{rule}\ \texttt{TC}\ \theta'(\overline{b})\ \Longrightarrow\ u_{i_0} \sim b_{i_0}$, where

$$\theta'(b_j) = \begin{cases} u_j & \text{, if } j \in \{i_1, \ldots, i_n\}\\ b_j & \text{, otherwise}\end{cases}$$

We can derive this rule, by combining Schemes CS1b and CS2b:

$$\dfrac{\dfrac{\dfrac{}{F_{\mathtt{TC}_i}\ \overline{u}^{i_n} \sim \theta(u_{i_0})}\ 2\textsc{b}}{\theta(u_{i_0}) \sim F_{\mathtt{TC}_i}\ \overline{u}^{i_n}}\ \textsc{Sym} \qquad \dfrac{\dfrac{\mathtt{TC}\ \theta'(\overline{b})}{F_{\mathtt{TC}_i}\ \theta'(\overline{b}^{i_n}) \sim \theta'(b_{i_0})}\ 1\textsc{b}}{F_{\mathtt{TC}_i}\ \overline{u}^{i_n} \sim b_{i_0}}\ \theta'}{\theta(u_{i_0}) \sim b_{i_0}}\ \text{ş}$$

The corresponding System F_C term has exactly the same structure: a local pattern match (encoding Scheme CS1b) against the dictionary of type ($\mathtt{TC}\ \theta'(\overline{b})$) in order to expose equality $F_{\mathtt{TC}_i}\ \overline{u}^{i_n} \sim b_{i_0}$, which is then combined with top-level axiom $\theta(u_{i_0}) \sim F_{\mathtt{TC}_i}\ \overline{u}^{i_n}$ (from Scheme CS2b) to produce $\theta(u_{i_0}) \sim b_{i_0}$.

Notice that, similarly to the Instance CHR, we do not actually prove $u_{i_0} \sim b_{i_0}$, but the refined $\theta(u_{i_0}) \sim b_{i_0}$.

B.2 Poly-kinded, Generic Type Projections

Even though we omitted kinds from our main presentation for brevity, it is quite straightforward to extend the system of Section 8.3 with kind checking (quite cumbersome though). More importantly, if we further extend the system with *kind polymorpism* (Yorgey et al., 2012) – which is also straightforward – there is no need to axiomatize the projection functions we presented in Section 8.3.5 for each data type. Instead, we can perform type projection generically, using only two user-defined kind-polymorphic type families L and R, along with two axioms:

$$\begin{array}{rcl}
\textbf{type}\ L & : & \forall \kappa_1\ \kappa_2.\ (a : \kappa_1) \rightarrow \kappa_2 \\
\textbf{type}\ R & : & \forall \kappa_1\ \kappa_2.\ (a : \kappa_1) \rightarrow \kappa_2 \\[4pt]
\textbf{axiom}\ \mathit{proj}_L & : & L\ ((u_1 : \kappa_2 \rightarrow \kappa_1)\ (u_2 : \kappa_2)) \sim u_1 \\
\textbf{axiom}\ \mathit{proj}_R & : & R\ ((u_1 : \kappa_2 \rightarrow \kappa_1)\ (u_2 : \kappa_2)) \sim u_2
\end{array}$$

For example, instead of the axioms $\mathit{Fst}\ (a, b) \sim a$ and $\mathit{Snd}\ (a, b) \sim b$, we can extract the first and the second component of a tuple type τ as follows:

$$\begin{array}{rcl}
\mathit{Fst}\ \tau & \equiv & R\ (L\ \tau) \\
\mathit{Snd}\ \tau & \equiv & R\ \tau
\end{array}$$

Figure B.1 Elaboration of Top-level Bindings

$\boxed{P; \Gamma \vdash_{\texttt{VAL}} \mathit{val} : \Gamma_v \rightsquigarrow \mathit{decl}}$ Value Binding Elaboration

$$\frac{\begin{array}{c}\Gamma \vdash_{\texttt{TM}} e : \tau \rightsquigarrow t \mid \Pi_1; \mathcal{E}_1 \\ P \vdash_{\texttt{E}} (\Pi_1, \mathcal{E}_1) \rightsquigarrow^{!} (\Pi_2, \mathcal{E}_2); \theta; \eta \qquad \overline{a} = \mathit{fuv}(\Pi_2, \mathcal{E}_2, \theta(\tau)) \\ \sigma = \forall \overline{a}.\ (\mathit{erase}(\mathcal{E}_2), \mathit{erase}(\Pi_2)) \Rightarrow \theta(\tau) \qquad \Gamma \vdash_{\texttt{TY}} \sigma \rightsquigarrow \upsilon\end{array}}{P; \Gamma \vdash_{\texttt{VAL}} (x = e) : [x : \sigma] \rightsquigarrow \textbf{let}\ x : \upsilon = \Lambda \overline{a}.\ \Lambda \mathcal{E}_2.\ \lambda \Pi_2.\ \theta(\eta(t))}\ \textsc{Val}$$

Furthermore, this approach is strictly more powerful than the projection functions of Section 8.3.5. Consider the following example:[2]

$$\begin{array}{l}\textbf{class}\ A\ a\ b \mid a \to b \\ \textbf{class}\ B\ a\ b \mid a \to b \\ \textbf{class}\ C\ a\ b \mid a \to b \\ \\ \textbf{instance}\ (\mathit{Functor}\ f, A\ a\ (f\ b), B\ b\ c) \Rightarrow C\ a\ c\end{array}$$

In the instance for C, c is determined by a, indirectly through b. Yet, $(f\ b)$ is not a type constructor application of the form $T\ \overline{a}$, so b cannot be extracted using a projection function of the form $\mathit{Proj}_i^T(\cdot)$. Using the polymorphic projection functions L and R though, we can easily derive the implied axiom from the above instance. We have the following implications:

$$\begin{array}{llllll}A\ a\ (f\ b) & \Longrightarrow & F_A\ a \sim f\ b & \Longrightarrow & b \sim R\ (F_A\ a) \\ B\ b\ c & \Longrightarrow & F_B\ b \sim c & \Longrightarrow & c \sim F_B\ b\end{array}$$

Thus, the derived axiom for F_C is the following:

$$\textbf{axiom}\ F_C\ a : F_C\ a \sim F_B\ (R\ (F_A\ a))$$

B.3 Elaboration of Top-level Value Bindings

Relation $P; \Gamma \vdash_{\texttt{VAL}} \mathit{val} : \Gamma_v \rightsquigarrow \mathit{decl}$ performs type inference (and elaboration into System F$_\mathrm{C}$) for top-level bindings. It is given by a single rule, presented in Figure B.1.[3]

To stay in line with the primary goal of functional dependencies, type improvement (Jones, 1995c), once we have inferred the type of the binding we

[2] Michiel Dehaeg, personal communication.
[3] $\mathit{fuv}(\cdot)$ computes the free unification variables of types and constraints.

perform simplification, via constraint entailment. Function $erase(\cdot)$ removes all evidence annotations from either class constraints Π or equality constraints $\mathcal{E}$.

Notice that, for simplicity, we have considered only top-level bindings without type signatures (just as we did for the basic system and our first extension) but it is extremely straightforward to allow explicit type annotations. A top-level binding $val \equiv (x :: \sigma) = e$ can be handled by the type-checking relation $P; \Gamma \vdash_{\texttt{TM}} e : \sigma \rightsquigarrow t$ we gave in Section 8.5.6:

$$\frac{P; \Gamma \vdash_{\texttt{TM}} e : \sigma \rightsquigarrow t \qquad \Gamma \vdash_{\texttt{TY}} \sigma \rightsquigarrow \upsilon}{P; \Gamma \vdash_{\texttt{VAL}} (x :: \sigma = e) : \Gamma_{\upsilon} \rightsquigarrow \mathbf{let}\ x : \upsilon = t}\ \textsc{ValAnn}$$

Index

Chapter 2
- K, 12
- $p \preceq v$, 18
- T, 12

Chapter 3
- M, 52
- Q, 39
- V, W, 50
- Δ, 39
- $[\![S]\!] = \overline{\vec{V}}$, 51
- $[\![\Gamma]\!] = \overline{\theta}$, 51
- $[\![\vec{p}]\!](\overline{\vec{V}}) = \langle\overline{\vec{V}}_c, \overline{\vec{V}}_u, \overline{\vec{V}}_\perp\rangle\rangle$, 51
- $[\![\vec{p}]\!]^\theta(\vec{V}) = M$, 51
- $\mathcal{C}\ \vec{p}\ v$, 41
- $\mathcal{D}\ \vec{p}\ v$, 41
- $\mathcal{U}\ \vec{p}\ v$, 41
- $patVecProc(\cdot, \cdot)$, 41
- τ_c, 50
- $\vdash_{\textsc{Sat}} w$, 41
- $a, b, a', b', \ldots$, 39
- $f, g, x, y, \ldots$, 34
- u, w, 39
- v, 39
- $kcon$, 42
- $tail$, 43
- $ucon$, 43
- c, 34
- G, 34
- p, q, 34
- K, 34
- $translate_c(\mathbb{C}) = \vec{p}$, 36
- $translate_g(\mathbb{G}) = \vec{p}$, 36
- $translate_{\vec{g}}(\vec{\mathbb{G}}) = \vec{p}$, 38
- $translate_p(\mathbb{P}) = \vec{p}$, 36
- $translate_{\vec{p}}(\vec{\mathbb{P}}) = \vec{p}$, 37
- e, 34
- $\mathscr{S}, \mathscr{C}, \mathscr{U}, \mathscr{D}$, 51
- S, C, U, D, 39
- C, 31
- D, 31
- U, 31
- $\mathbb{C}$, 34
- $\mathbb{G}$, 34
- $\mathbb{P}$, 34
- T, 39
- τ, 39
- $\vdash_{\text{V}} V : \tau_c$, 50
- $\Gamma \vdash_{\text{E}} e : \tau$, 50
- Γ, 39

Chapter 4
- $map_{\mathcal{R}}$, 64
- $\textsc{Match}(\vec{p},\ S) \Downarrow \mathcal{R}$, 67
- $\textsc{Match}(\vec{p},\ v) \Downarrow \mathcal{R}$, 65
- $\mathcal{R}$, 64
- e', 81
- p, q, 67
- u, w, 67

Chapter 5
- Section 5.2
  - V, 93

$\mathit{erase}(t)$, 94
$t_1 \longrightarrow t_2$, 92
$\Gamma \vdash_{\mathtt{TM}} t : \upsilon$, 92
$\Gamma \vdash_{\mathtt{TY}} \upsilon$, 92
υ, 92
Γ, 92
t, 92
Section 5.3
E, 100
$\Gamma \vdash_{\mathtt{TM}} e : \tau \rightsquigarrow t \mid E$, 101
$\mathit{erase}(t)$, 106
$\vdash_{\mathtt{GEN}} e : \sigma \rightsquigarrow t$, 101
$\Gamma \models \sigma_0 \preceq \sigma$, 104
$\Gamma \vdash_{\mathtt{TM}} e : \sigma \rightsquigarrow t$, 99
$\Gamma \vdash_{\mathtt{TY}} \sigma \rightsquigarrow \upsilon$, 99
$\mathit{unify}(E) = \theta_\bot$, 101
τ, 98
σ, 98
Γ, 99
θ, 100
e, 98
Section 5.4
f_n, 110
p_n, 111
t, 110
ϕ, 111
Chapter 6
$\mathtt{TC}$, 122
Π, 123
$\mathcal{Q}$, 123
$\mathcal{S}$, 123
$\mathcal{C}$, 123
$\mathcal{A}$, 123
π, 122
cls, 122
Q, 122
S, 122
C, 122
decl, 122
η, 128
$P; \Gamma \models \sigma_1 \preceq \sigma_2$, 134
P, 122
$\mathit{proj}^i_{\mathtt{TC}}(d)$, 127
ins, 122
$\mathbb{E}$, 132
τ, 122
$\mathit{mponens}^*(\overline{a}, \mathcal{A}_S, \mathcal{C}_L) = (\mathcal{C}'_L, \mathbb{E})$, 132
$\mathit{mponens}(\overline{a}, \mathcal{A}, \Pi) = (\mathcal{C}_L, \mathbb{E})$, 132
$\|Q\| = \mathbb{N}$, 136
$\|\tau\| = \mathbb{N}$, 136
$\mathit{occ}_a(Q) = \mathbb{N}$, 136
$\mathit{occ}_a(\tau) = \mathbb{N}$, 136
σ, 122
pgm, 122
ρ, 122
$\textsc{ScClosure}(\overline{a}, P) = (\mathcal{A}, \mathbb{E})$, 132
$\mathit{elab}_{\mathtt{CT}}(Q) = \upsilon$, 129
$P; \Gamma \vdash_{\mathtt{INS}} \mathit{ins} : P' \rightsquigarrow \mathit{decl}$, 133
$\Gamma \vdash_{\mathtt{TM}} e : \tau \rightsquigarrow t \mid \mathcal{C} ; E$, 128
$\mathit{elab}_{\mathtt{TY}}(\sigma) = \upsilon$, 129
$P; \Gamma \vdash_{\mathtt{VAL}} \mathit{val} : \Gamma' \rightsquigarrow \mathit{decl}$, 133
$P; \Gamma \models t : Q$, 125
$\overline{a}; \mathcal{A} \models \mathcal{C}_1 \rightsquigarrow \mathcal{C}_2; \eta$, 130
$\overline{a}; \mathcal{A} \models \mathcal{Q} \rightsquigarrow \mathcal{C}; \eta$, 130
$\overline{a}; P; \Gamma \vdash_{\mathtt{TM}} e : \sigma \rightsquigarrow t$, 134
$\Gamma \vdash_{\mathtt{CLS}} \mathit{cls} : P_S; \Gamma_c \rightsquigarrow \overline{\mathit{decl}}$, 126
$\Gamma \vdash_{\mathtt{CT}} Q \rightsquigarrow \upsilon$, 124
$P; \Gamma \vdash_{\mathtt{INS}} \mathit{ins} : P_i \rightsquigarrow \mathit{decl}$, 126
$\vdash_{\mathtt{PGM}} \mathit{pgm} : P; \Gamma \rightsquigarrow \overline{\mathit{decl}}$, 127
$P; \Gamma \vdash_{\mathtt{TM}} e : \sigma \rightsquigarrow t$, 124
$\Gamma \vdash_{\mathtt{TY}} \sigma \rightsquigarrow \upsilon$, 123
$P; \Gamma \vdash_{\mathtt{VAL}} \mathit{val} : \Gamma' \rightsquigarrow \mathit{decl}$, 126
$\mathit{unamb}(S)$, 137
$\mathit{unify}(\overline{a}; E) = \theta_\bot$, 130
$\mathit{term}(S)$, 136
Γ, 122
val, 122
d, 122
e, 122
Section 6.1
Γ, 119
$t_1 \longrightarrow t_2$, 120

$\Gamma \vdash_{\texttt{D}} decl : \Gamma'$, 120
$\Gamma \vdash_{\texttt{P}} p \to t : \upsilon_1 \to \upsilon_2$, 120
$\Gamma \vdash_{\texttt{TM}} t : \upsilon$, 119
$\Gamma \vdash_{\texttt{TY}} \upsilon$, 119
$decl$, 118
pgm, 118
p, 118
υ, 118
t, 118

Chapter 7
Q, 144
P, 144
$elab_{\texttt{CT}}(Q) = \upsilon$, 152
$\overline{a}; P \models \mathcal{C}_1 \rightsquigarrow \mathcal{C}_2; \eta$, 154
$P; \Gamma \models t : Q$, 147, 149
$head(Q) = \pi$, 157
$\overline{a}; [\mathcal{Q}] \models \Pi \rightsquigarrow \mathcal{C}; \theta; \eta$, 154
$\Gamma; t : [Q] \models t' : \pi \rightsquigarrow \mathcal{C}$, 149
$\overline{a}; P \models [\mathcal{Q}] \rightsquigarrow \mathcal{C}; \eta$, 154
$P; \Gamma \models t : [Q]$, 149
$\Gamma \vdash_{\texttt{CT}} Q \rightsquigarrow \upsilon$, 146
$term(Q)$, 157
$\overline{a} \vdash_{\texttt{UNAMB}} Q$, 151
$unamb(Q)$, 151

Chapter 8
Π, 189
α, β, 189
$\text{BINDS}(\mathbb{E}) = \Gamma; P$, 190
cls, 172
Q, 172
$P \vdash_{\texttt{E}} Q \rightsquigarrow C; \theta; \eta$, 195
$P \vdash_{\texttt{E}} \mathcal{C} \rightsquigarrow^* \mathcal{C}; \theta; \eta$, 195
$\overline{a}; \overline{\pi} \vdash_{\texttt{D}} \bullet \rightsquigarrow^! \theta$, 180
$det(\overline{a}, \overline{\pi}) = \theta$, 180
$\overline{a}; \overline{\pi} \vdash_{\texttt{D}} \theta_1 \rightsquigarrow \theta_2$, 180
$\Gamma \vdash_{\texttt{CLS}} cls \rightsquigarrow \overline{decl} \mid \Gamma_c$, 196
$P; \Gamma \vdash_{\texttt{INS}} ins \rightsquigarrow \overline{decl} \mid P_i$, 197
$\Gamma \vdash_{\texttt{TM}} e : \tau \rightsquigarrow t \mid \Pi; \mathcal{E}$, 191
$P; \Gamma \vdash_{\texttt{VAL}} val : \Gamma_v \rightsquigarrow decl$, 195
$P_{inst} \vdash_{\texttt{SC}} (\texttt{TC}\ \overline{u}) \rightsquigarrow (\overline{\tau}, \overline{t}, \overline{\gamma})$, 198
η, 189
fd, 172
π, 172
$elab_{\texttt{CC}}(\pi) = \upsilon$, 192
$elab_{\texttt{EQ}}(\phi) = \psi$, 192
$elab_{\texttt{PT}}(P) = \Gamma$, 192
$elab_{\texttt{TE}}(\Gamma) = \Gamma'$, 192
$elab_{\texttt{TY}}(\sigma) = \upsilon$, 192
$I; \Gamma \models S$, 179
ϕ, 172
$Proj_i^T(\cdot)$, 181
$I; \Gamma \models \sigma_0 \preceq \sigma$, 203
$I; \Gamma \vdash_{\texttt{CLS}} cls : I_c; \Gamma_c$, 179
$\Gamma \vdash_{\texttt{CT}} Q$, 178
$I_c; I_i; \Gamma \vdash_{\texttt{INS}} ins : I$, 179
$\vdash_{\texttt{PGM}} pgm$, 180
$I; \Gamma \vdash_{\texttt{TM}} e : \sigma$, 177
$\Gamma \vdash_{\texttt{TY}} \sigma$, 178
$I; \Gamma_1 \vdash_{\texttt{VAL}} val : \Gamma_2$, 179
$unambig(\overline{b}, \overline{a}, \overline{\pi})$, 180
$fixed(\tau) = \overline{a}$, 203
P, 189
$S_I \hookrightarrow P_{ax}$, 197
ins, 172
$\Pi \Downarrow \mathbb{E}$, 190
$\mathbb{E}$, 190
$\mathcal{C}$, 189
$\mathcal{E}$, 189
$\mathcal{Q}$, 189
I, 177
$closure(\overline{a}, \overline{\pi})$, 180
τ, 172
$P \vdash_{\texttt{U}} \omega : \tau_1 \sim \tau_2 \rightsquigarrow \mathcal{E}; \theta; \eta$, 194
$P; \Gamma \vdash_{\texttt{TM}} e : \sigma \rightsquigarrow t$, 197
θ, 189
F, 172
u, 172
$P \vdash_{\texttt{R}} \tau \rightsquigarrow \tau'; \gamma$, 193

Section 8.4
Γ, 184
$compat(\phi_1, \phi_2)$, 188
$decl$, 186
$t_1 \longrightarrow t_2$, 187
$\Gamma \vdash_{\texttt{CO}} \gamma : \psi$, 185

$\Gamma_1 \vdash_{\mathtt{D}} decl : \Gamma_2$, 187
$\Gamma \vdash_{\mathtt{P}} p \rightarrow t : \upsilon_1 \rightarrow \upsilon_2$, 186
$\Gamma \vdash_{\mathtt{PR}} \psi$, 183
$\Gamma \vdash_{\mathtt{TM}} t : \upsilon$, 184
$\Gamma \vdash_{\mathtt{TY}} \upsilon$, 183
γ, 183
p, 185
ω, 184
ψ, 183
υ, 182
g, 184
t, 184
u, 186
Chapter 9
$\mathit{ctxProj}^{i}_{\mathtt{TC}}(d)$, 218
$\textsc{InvScClosure}(\overline{a}, P) = (\mathcal{A}, \mathbb{E})$, 219
P, 216

Bibliography

Andreoli, J.-m. (1992). Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2:297–347.

Augustsson, L. (1985). Compiling pattern matching. In *Proceedings of the 1985 Conference on Functional Programming and Computer Architecture.*

Augustsson, L. and Petersson, K. (1994). Silly type families. `http://web.cecs.pdx.edu/~sheard/papers/silly.pdf`.

Bird, R. S. and Meertens, L. G. L. T. (1998). Nested datatypes. In *MPC '98*, pages 52–67. Springer.

Bjørner, N. S. (1994). Minimal typing derivations. In *In ACM SIGPLAN Workshop on ML and its Applications*, pages 120–126.

Böhm, C. and Berarducci, A. (1985). Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39(0):135 – 154.

Brady, E. (2013). Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593.

Burstall, R. M., MacQueen, D. B., and Sannella, D. T. (1980). HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages 136–143, New York, NY, USA. ACM.

Chakravarty, M. M. T., Keller, G., and Jones, S. P. (2005a). Associated type synonyms. *SIGPLAN Not.*, 40(9):241–253.

Chakravarty, M. M. T., Keller, G., Jones, S. P., and Marlow, S. (2005b). Associated types with class. *SIGPLAN Not.*, 40(1):1–13.

Chauhan, S., Kurur, P. P., and Yorgey, B. A. (2016). How to twist pointers without breaking them. In *Haskell 2016*, pages 51–61. ACM.

Cheney, J. and Hinze, R. (2003). First-class phantom types. Technical report, Cornell University.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):354–363.

Claessen, K., Johansson, M., Rosén, D., and Smallbone, N. (2013). Automating inductive proofs using theory exploration. In Bonacina, M. P., editor, *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 392–406. Springer.

Coquand, T. (1992). Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs.*

Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *POPL '82*, pages 207–212. ACM.

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. (2006). Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388.

Duck, G. J., Peyton-Jones, S., Stuckey, P. J., and Sulzmann, M. (2004). Sound and decidable type inference for functional dependencies. In *TOPLAS*, volume 2986 of *Lecture Notes in Computer Science*, pages 49–63. Springer.

Dunfield, J. (2007a). Refined typechecking with Stardust. In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*, PLPV '07, pages 21–32, New York, NY, USA. ACM.

Dunfield, J. (2007b). *A Unified System of Type Refinements.* PhD thesis, Carnegie Mellon University. CMU-CS-07-129.

Eisenberg, R. A., Vytiniotis, D., Peyton Jones, S., and Weirich, S. (2014). Closed type families with overlapping equations. In *POPL '14*.

Erwig, M. and Peyton Jones, S. (2000). Pattern guards and transformational patterns. In *Proceedings of the 2000 Haskell Symposium.* ACM.

Frühwirth, T. W. (1995). Constraint handling rules. In *Selected Papers from Constraint Programming: Basics and Trends*, pages 90–107. Springer-Verlag.

Fu, P., Komendantskaya, E., Schrijvers, T., and Pond, A. (2016). Proof relevant corecursive resolution. In *FLOPS 2016*, pages 126–143. Springer.

Garrigue, J. and Normand, J. L. (2011). Adding GADTs to OCaml: the direct approach. In *Workshop on ML.*

Garrigue, J. and Normand, J. L. (2015). GADTs and exhaustiveness: Looking for the impossible. In *ML Family/OCaml.*

Gentzen, G. (1935). Untersuchungen über das logische schließen. i. *Mathematische Zeitschrift*, 39(1):176–210.

Girard, J.-Y. (1972). *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, PhD thesis, Université Paris VII.

Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., and Lumsdaine, A. (2006). Concepts: Linguistic support for generic programming in c++. *SIGPLAN Not.*, 41(10):291–310.

Hall, C. V., Hammond, K., Peyton Jones, S. L., and Wadler, P. L. (1996). Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2).

Hallgren, T. (2000). Fun with functional dependencies. In *Proc. of the Joint CS/CE Winter Meeting*.

Harrop, R. (1956). On disjunctions and existential statements in intuitionistic systems of logic. *Mathematische Annalen*, 132(4):347–361.

Henderson, F., Conway, T., Somogyi, Z., Jeffery, D., Schachte, P., Taylor, S., and Speirs, C. (1996). The mercury language reference manual. Technical report.

Hilbert, D. and Bernays, P. (1934). *Grundlagen der Mathematik*, volume 1 of *Die Grundlehren der mathematischen Wissenschaften*. Verlag von Julius Springer.

Hindley, R. (1969). The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60.

Hinze, R. (2000). Perfect trees and bit-reversal permutations. *JFP*, 10(3):305–317.

Hinze, R. (2010). Adjoint folds and unfolds: Or: Scything through the thicket of morphisms. In *MPC'10*, pages 195–228. Springer.

Hinze, R. and Peyton Jones, S. (2000). Derivable type classes. In *Proceedings of the Fourth Haskell Workshop*, pages 227–236. Elsevier Science.

Jaskelioff, M. (2011). Monatron: an extensible monad transformer library. In *IFL'08*, pages 233–248, Berlin, Heidelberg. Springer.

Johann, P. and Ghani, N. (2008). Foundations for structured programming with gadts. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 297–308, New York, NY, USA. ACM.

Jones, M. (2010). *The Habit Programming Language: The Revised Preliminary Report.*

Jones, M. P. (1992). A theory of qualified types. In Krieg-Brückner, B., editor, *ESOP '92*, volume 582 of *LNCS*, pages 287–306. Springer Berlin Heidelberg.

Jones, M. P. (1993). A system of constructor classes: Overloading and implicit higher-order polymorphism. In *FPCA '93*, pages 52–61. ACM.

Jones, M. P. (1995a). Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, pages 97–136. Springer.

Jones, M. P. (1995b). *Qualified Types: Theory and Practice.* Cambridge University Press.

Jones, M. P. (1995c). Simplifying and improving qualified types. In *FPCA '95*, pages 160–169. ACM.

Jones, M. P. (2000). Type classes with functional dependencies. In *Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*. Springer.

Jones, M. P. and Diatchki, I. S. (2008). Language and program design for functional dependencies. *SIGPLAN Not.*, 44(2):87–98.

Jones, S. P., Jones, M., and Meijer, E. (1997). Type classes: an exploration of the design space. In *Proceedings of the 1997 Haskell Workshop*. ACM.

Karachalias, G., Schrijvers, T., Vytiniotis, D., and Jones, S. P. (2015). GADTs meet their match: Pattern-matching warnings that account for GADTs, guards, and laziness. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 424–436, New York, NY, USA. ACM.

Kiselyov, O., Lämmel, R., and Schupke, K. (2004). Strongly typed heterogeneous collections. In *Haskell '04*, pages 96–107. ACM.

Kmett, E. A. (2017). The constraint package. `https://hackage.haskell.org/package/constraints-0.9.1`.

Koot, R. and Hage, J. (2015). Type-based exception analysis for non-strict higher-order functional languages with imprecise exception semantics. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, PEPM '15, pages 127–138, New York, NY, USA. ACM.

Kowalski, R. (1974). Predicate logic as programming language. In *Proceedings of IFIP '74*, pages 569 – 574, North Holland.

Krishnaswami, N. R. (2009). Focusing on pattern matching. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 366–378, New York, NY, USA. ACM.

Lämmel, R. and Peyton Jones, S. (2003). Scrap your boilerplate: A practical design pattern for generic programming. *SIGPLAN Not.*, 38(3):26–37.

Lämmel, R. and Peyton Jones, S. (2005). Scrap your boilerplate with class: Extensible generic functions. *SIGPLAN Not.*, 40(9):204–215.

Laville, A. (1991). Comparison of priority rules in pattern matching and term rewriting. *J. Symb. Comput.*, 11(4):321–347.

Le Fessant, F. and Maranget, L. (2001). Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming.*

Liang, C. and Miller, D. (2009). Focusing and polarization in linear, intuitionistic, and classical logics. *Theor. Comput. Sci.*, 410(46):4747–4768.

Lindahl, T. and Sagonas, K. (2004). Detecting software defects in telecom applications through lightweight static analysis: A war story. In Chin, W.-N., editor, *Programming Languages and Systems*, pages 91–106, Berlin, Heidelberg. Springer Berlin Heidelberg.

Maranget, L. (1992). Compiling lazy pattern matching. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 21–31, New York, NY, USA. ACM.

Maranget, L. (2007). Warnings for pattern matching. *Journal of Functional Programming*, 17:387–421.

Maranget, L. (2008). Compiling pattern matching to good decision trees. In *Proceedings of the ACM Workshop on ML.*

Maranget, L. and Para, P. (1994). Two techniques for compiling lazy pattern matching. Technical report.

Martelli, A. and Montanari, U. (1982). An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282.

The Coq development team (2004). *The Coq proof assistant reference manual.* LogiCal Project. Version 8.0.

McBride, C. and McKinna, J. (2004). The view from the left. *Journal of Functional Programming*, 14(1):69–111.

Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. (1989). Uniform proofs as a foundation for logic programming. Technical report.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

Mitchell, N. and Runciman, C. (2008). Not all patterns, but enough: An automatic verifier for partial but sufficient pattern matching. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 49–60, New York, NY, USA. ACM.

Morris, J. G. (2014). A simple semantics for haskell overloading. *SIGPLAN Not.*, 49(12):107–118.

Morris, J. G. and Eisenberg, R. A. (2017). Constrained type families. *Proc. ACM Program. Lang.*, 1(ICFP):42:1–42:28.

Musser, D. R. and Stepanov, A. A. (1989). Generic programming. In Gianni, P., editor, *Symbolic and Algebraic Computation*, pages 13–25, Berlin, Heidelberg. Springer Berlin Heidelberg.

Norell, U. (2007). *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Norell, U. (2009). Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA. ACM.

Oliveira, B. C., Schrijvers, T., Choi, W., Lee, W., and Yi, K. (2012). The implicit calculus: A new foundation for generic programming. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 35–44, New York, NY, USA. ACM.

Peano, G. (1889). *Arithmetices principia: nova methodo exposita.* Fratres Bocca.

Peyton Jones, S. (1997). Bulk types with class. In *Proceedings of the Second Haskell Workshop.*

Peyton Jones, S. (2003). *Haskell 98 Language and Libraries: The Revised Report.* Journal of functional programming. Cambridge University Press.

Peyton Jones, S., Vytiniotis, D., Weirich, S., and Washburn, G. (2006). Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61.

Pickering, M., Érdi, G., Peyton Jones, S., and Eisenberg, R. A. (2016). Pattern synonyms. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, pages 80–91, New York, NY, USA. ACM.

Pierce, B. C. (2002). *Types and Programming Languages.* The MIT Press, 1st edition.

Ramsey, N., Dias, J. a., and Peyton Jones, S. (2010). Hoopl: A modular, reusable library for dataflow analysis and transformation. *SIGPLAN Not.*, 45(11):121–134.

Reynolds, J. C. (1974). Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 408–423, London, UK, UK. Springer-Verlag.

Reynolds, J. C. (1983). Types, abstraction, and parametric polymorphism. In Mason, R., editor, *Information Processing 83*, pages 513–523, North Holland, Amsterdam.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41.

Rondon, P. M., Kawaguci, M., and Jhala, R. (2008). Liquid types. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA. ACM.

Sagonas, K., Silva, J., and Tamarit, S. (2013). Precise explanation of success typing errors. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13, pages 33–42, New York, NY, USA. ACM.

Schrijvers, T. and Oliveira, B. C. (2011). Monads, zippers and views: Virtualizing the monad stack. *SIGPLAN Not.*, 46(9):32–44.

Schrijvers, T., Oliveira, B. C. d. S., and Wadler, P. (2017). Cochis: Deterministic and coherent implicits. Report CW 705, KU Leuven.

Schrijvers, T., Peyton Jones, S., Chakravarty, M., and Sulzmann, M. (2008). Type checking with open type functions. In *ICFP '08*, pages 51–62. ACM.

Schrijvers, T., Peyton Jones, S., Sulzmann, M., and Vytiniotis, D. (2009). Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 341–352, New York, NY, USA. ACM.

Schrijvers, T. and Sulzmann, M. (2008). Unified type checking for classes and type families.

Schrijvers, T., Sulzmann, M., Peyton Jones, S., and Chakravarty, M. (2007). Towards open type functions for Haskell. In *IFL '07*, pages 233–251.

Sekar, R. C., Ramesh, R., and Ramakrishnan, I. V. (1995). Adaptive pattern matching. *SIAM J. Comput.*, 24(6):1207–1234.

Serrano, A., Hage, J., and Bahr, P. (2015). Type families with class, type classes with family. *SIGPLAN Not.*, 50(12):129–140.

Sestoft, P. (1996). ML pattern match compilation and partial evaluation. In Danvy, O., Glück, R., and Thiemann, P., editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 446–464. Springer Berlin Heidelberg.

Shärli, N., Ducasse, S., Nierstrasz, O., and Black, A. (2002). Traits: Composable units of behavior. Technical report.

Sheard, T. (2004). Languages of the future. In *In OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 116–119. ACM Press.

Silberschatz, A., Korth, H., and Sudarshan, S. (2006). *Database Systems Concepts*. McGraw-Hill, Inc., 5 edition.

Sonnex, W., Drossopoulou, S., and Eisenbach, S. (2012). Zeno: An automated prover for properties of recursive data structures. pages 407–421. Springer-Verlag Berlin.

Sozeau, M. and Oury, N. (2008). First-class type classes. In *TPHOLs 2008*, pages 278–293.

Spivey, M. (2017). Faster coroutine pipelines. *Proc. ACM Program. Lang.*, 1(ICFP):5:1–5:23.

Stolarek, J., Peyton Jones, S., and Eisenberg, R. A. (2015). Injective type families for Haskell. *SIGPLAN Not.*, 50(12):118–128.

Stuckey, P. J. and Sulzmann, M. (2005). A theory of overloading. *ACM Trans. Program. Lang. Syst.*, 27(6):1216–1269.

Sulzmann, M., Chakravarty, M. M. T., Jones, S. P., and Donnelly, K. (2007a). System F with type equality coercions. In *TLDI '07*. ACM.

Sulzmann, M., Duck, G. J., Peyton-Jones, S., and Stuckey, P. J. (2007b). Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1):83–129.

Tarski, A. (1933). The concept of truth in the languages of the deductive sciences (Polish). *Prace Towarzystwa Naukowego Warszawskiego, Wydzial III Nauk Matematyczno-Fizycznych 34*.

Trifonov, V. (2003). Simulating quantified class constraints. In *Haskell '03*, pages 98–102. ACM.

Urzyczyn, P. (1997). Inhabitation in typed lambda-calculi (a syntactic approach). In *Proceedings of the Third International Conference on Typed Lambda Calculi and Applications*, TLCA '97, pages 373–389, London, UK, UK. Springer-Verlag.

Vazou, N., Seidel, E. L., Jhala, R., Vytiniotis, D., and Peyton-Jones, S. (2014). Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA. ACM.

Vytiniotis, D., Peyton Jones, S., and Schrijvers, T. (2010). Let should not be generalized. In *TLDI '10*, pages 39–50. ACM.

Vytiniotis, D., Peyton jones, S., Schrijvers, T., and Sulzmann, M. (2011). Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412.

Vytiniotis, D., Weirich, S., and Peyton Jones, S. (2008). Fph: First-class polymorphism for haskell. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 295–306, New York, NY, USA. ACM.

Wadler, P. (1987a). Efficient compilation of pattern matching. In Peyton Jones, S., editor, *The implementation of functional programming languages*, pages 78–103. Prentice Hall.

Wadler, P. (1987b). Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 307–313, New York, NY, USA. ACM.

Wadler, P. (1989). Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA. ACM.

Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, pages 60–76. ACM.

Weirich, S., Hsu, J., and Eisenberg, R. A. (2013). System fc with explicit kind equality. *SIGPLAN Not.*, 48(9):275–286.

Weirich, S., Voizard, A., de Amorim, P. H. A., and Eisenberg, R. A. (2017). A specification for dependent types in haskell. *Proc. ACM Program. Lang.*, 1(ICFP):31:1–31:29.

Weirich, S., Vytiniotis, D., Peyton Jones, S., and Zdancewic, S. (2011). Generative type abstraction and type-level computation. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 227–240, New York, NY, USA. ACM.

Wells, J. B. (1993). Typability and type checking in the second-order lambda-calculus are equivalent and undecidable. Technical report, Boston, MA, USA.

Wright, A. and Felleisen, M. (1994). A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94.

Xi, H. (1998a). Dead code elimination through dependent types. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, PADL '99, pages 228–242, London, UK. Springer-Verlag.

Xi, H. (1998b). *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University.

Xi, H. (2003). Dependently typed pattern matching. *Journal of Universal Computer Science*, 9:851–872.

Xi, H., Chen, C., and Chen, G. (2003). Guarded recursive datatype constructors. *SIGPLAN Not.*, 38(1):224–235.

Xu, D. N. (2006). Extended static checking for Haskell. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 48–59, New York, NY, USA. ACM.

Xu, D. N., Peyton Jones, S., and Claessen, K. (2009). Static contract checking for Haskell. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 41–52, New York, NY, USA. ACM.

Yorgey, B. A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., and Magalhães, J. P. (2012). Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA. ACM.

List of Publications

Papers at International Conferences and Symposia

Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis and Simon Peyton Jones (2015). GADTs Meet Their Match: Pattern-matching Warnings That Account for GADTs, Guards, and Laziness. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, pp. 424–436, Vancouver, BC, Canada, August 31-September 2, 2015.

Georgios Karachalias and Tom Schrijvers (2017). Elaboration on Functional Dependencies: Functional Dependencies Are Dead, Long Live Functional Dependencies! In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell '17, pp. 133–147, Oxford, UK, September 7–8, 2017.

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira and Philip Wadler (2017). Quantified Class Constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell '17, pp. 148–161, Oxford, UK, September 7–8, 2017.

Amr Hany Saleh, Georgios Karachalias, Matija Pretnar, and Tom Schrijvers (2018). Explicit Effect Subtyping. In *Proceedings of the 27th European Symposium on Programming*, ESOP '18, Thessaloniki, Greece, April 16–19, 2018.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DTAI
Celestijnenlaan 200A box 2402
B-3001 Leuven
georgios.karachalias@cs.kuleuven.be
https://dtai.cs.kuleuven.be