PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/92736

Please be advised that this information was generated on 2017-10-25 and may be subject to change.
Dynamic Typing in Type-Driven Programming
Promotor:
  prof. dr. dr.h.c. ir. M.J. Plasmeijer

Copromotor:
  dr. P.M. Achten

Manuscriptcommissie:
  prof. dr. J.H. Geuvers
  prof. dr. S.D. Swierstra, Universiteit Utrecht
  prof. dr. F.H. Henglein, University of Copenhagen, Denmark

This research is supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organisation for Scientific Research (NWO) and partly funded by the Ministry of Economic Affairs, Agriculture and Innovation (project number 07729).

Printed by Gildeprint Drukkerijen, Enschede.

ISBN 978-94-6108-279-4
Acknowledgements

The last four years have been quite a ride. It involved hard work, several unproductive Friday afternoons, a healthy dose of frustration at times, and enjoyable trips abroad. Above all, it was one that I would not want to have missed and could not have completed without the help of others.

I am ever so grateful to my promotor Rinus for inviting me to come to Nijmegen in the first place. His enthusiasm about functional programming is contagious and he has always been very caring and welcoming whenever I desired to discuss something. Thankfully, he never shied away from giving his honest opinion about any idea I had. I especially would like to thank him for trusting me with the freedom to perform any research that sparked my interest.

Also, I am very thankful to my copromotor Peter for always being ready to answer any of my (little) questions and for the many fruitful discussions. He has given me essential pieces of advice while searching for the right direction of my research. His broad knowledge of functional programming has always been inspiring and his scrupulous evaluation of both code and text (and their layout) were a tremendous help to me.

I would like to thank all of my co-authors, Alexey, Bastiaan, Johan, John, Pedro, Pieter, Stefan, and Wouter, who have all inspired and supported me in my research. An additional thanks to John for his elaborate explanations of the details involved in the compilation of both Clean and Haskell, and to Wouter for teaching me everything I know about Agda.

Since writing a thesis is a solo venture that can be quite lonesome at times, having people around you who are in the same boat are ever so important not only to bounce ideas but also to vent any common frustrations. I am thankful to my roommates Bas, Freek, and Maarten for a fun and relaxed working environment and also to Bernard, Fabian, and Rody down the hall for the occasional conversation and enjoyable drinks. An additional thanks to Bas for helping me with the design of the cover of my thesis. I am also thankful to Doaitse for providing me with a desk to work in Utrecht whenever I pleased, and my roommates Alex and Bastiaan in Utrecht for the joyful atmosphere.

In my personal life there have also been a number of people who have supported me over the years. I thank my friends Arjan, Dennis, Jeffrey, Marc, and everyone from de Vlietstraat for showing interest in the progress of my research and for the many drinks we had whilst talking about (everyday) life. Also, Eelco has been of great importance by making me run and sweat intensively every week while playing squash.
I would like to thank my parents for their infinite support, despite the fact that I was not always able to explain the exact details of my research. Also, I am thankful to my brothers David and Fabian for being courageous enough to stand by me as *paranimfen*. Finally, I could not have been happier with Johanneke, she supports me in everything that I do and has been there for me all along.
Contents

1 Introduction
  1.1 Dynamic typing .............................. 1
  1.2 Type-driven programming ...................... 3
  1.3 Outline of this thesis ........................ 4
  1.4 Origin of chapters ........................... 5

I Functional languages

2 Exchanging sources between Clean and Haskell
  2.1 Introduction .................................. 9
  2.2 Modules ...................................... 11
    2.2.1 Module systems .......................... 11
    2.2.2 Compilation strategies .................. 12
    2.2.3 Mixed compilation ....................... 12
  2.3 Functions ................................... 13
  2.4 Macros ...................................... 16
  2.5 Newtypes .................................... 17
  2.6 Type classes ................................ 18
  2.7 Uniqueness typing ............................ 22
  2.8 Monads ....................................... 24
  2.9 Records ..................................... 26
  2.10 Arrays ...................................... 28
  2.11 Dynamic typing ............................. 31
  2.12 Generic functions ........................... 32
  2.13 Performance ................................ 34
  2.14 Related work ................................. 35
  2.15 Conclusion .................................. 36

II The necessity of dynamic typing

3 A lightweight approach to datatype-generic rewriting
  3.1 Introduction .................................. 41
  3.2 Representing rewrite rules .................... 43
    3.2.1 Intensional representations ............. 43
    3.2.2 Extensional representations ............. 45
## 3.3 Datatype-generic rewriting
- 3.3.1 Representing the structure of datatypes
- 3.3.2 Making terms rewritable

## 3.4 Generic equality

## 3.5 Matching and substituting
- 3.5.1 Typed metavariables
- 3.5.2 Generic patterns
- 3.5.3 Generic substitutions
- 3.5.4 Generic pattern matching

## 3.6 Synthesising rewrite rules
- 3.6.1 Generic sampling
- 3.6.2 Generic diff
- 3.6.3 Generic synthesis

## 3.7 Detecting ill-formed rewrite rules

## 3.8 Guarded rewriting

## 3.9 A case study: solving arithmetic equations

## 3.10 Performance
- 3.10.1 Turning propositions into disjunctive normal form
- 3.10.2 Solving arithmetic equations

## 3.11 Related work

## 3.12 Conclusion

### 4 Dynamic typing and generalised algebraic datatypes
- 4.1 Introduction
- 4.2 Preliminaries
- 4.3 Motivation
- 4.4 Semantics
- 4.5 Related work
- 4.6 Conclusion

### III Accommodating dynamic typing
- 5 Ad-hoc polymorphism and dynamic typing
## 5.3 Polymorphism

- **5.3.2 Polymorphism** ........................................... 104
- **5.3.3 Type dependencies** .................................... 105
- **5.3.4 Type codes** ........................................... 106

### 5.4 Dynamic typing in ad-hoc polymorphism

- **5.4.1 Container datatypes** ................................. 107
- **5.4.2 Dynamic dictionary composition** .................. 110

### 5.5 Ad-hoc polymorphism in dynamic typing

- **5.5.1 Dictionary-passing types** ............................. 113
- **5.5.2 Type code extension** ................................ 114

### 5.6 Type class extensions

- **5.6.1 Multi-parameter type classes** ...................... 116
- **5.6.2 Flexible contexts** .................................... 119
- **5.6.3 Flexible instances** .................................... 122

### 5.7 Related work

- ............................. 123

### 5.8 Conclusion

- ............................................................. 124

## 6 Embedding polymorphic dynamic typing

### 6.1 Introduction

- ............................................................. 125

### 6.2 Embedding in a functional language

- **6.2.1 Monomorphic dynamic typing** ..................... 127
- **6.2.2 Difficulties** ........................................ 129

### 6.3 Embedding in a dependently-typed language

- **6.3.1 Monomorphic dynamic typing** ..................... 130
- **6.3.2 Polymorphic dynamic typing** ...................... 131

### 6.4 Instance-of algorithm

- **6.4.1 Substitution** ........................................ 136
- **6.4.2 Accumulating instance-of function** ............... 137

### 6.5 Cast

- ............................................................. 139
- **6.5.1 Alignment** ........................................... 140
- **6.5.2 Instantiation** ........................................ 142
- **6.5.3 Putting it all together** .............................. 144

### 6.6 Related work

- ............................................................. 144

### 6.7 Conclusion

- ............................................................. 146

### 6.8 Proof prelude

- ............................................................. 147

### 6.9 Proofs regarding instance-of algorithm

- **6.9.1 Substitution** ........................................ 148
- **6.9.2 Accumulating instance-of function** ............... 149

### 6.10 Proofs regarding cast

- **6.10.1 Alignment** ........................................... 153
- **6.10.2 Instantiation** ........................................ 154

## Bibliography

- ............................................................. 155

## Summary

- ............................................................. 165

## Samenvatting

- ............................................................. 167

## Curriculum vitae

- ............................................................. 169
1 Introduction

It is human nature to make errors, but we rather not see this reflected in software end-products. An established approach to detecting many of the errors in a program is to use a programming language that is typed. In such languages, properties of operations and values are described by types. This allows a type system to verify that a program is well typed.

Preferably, verification takes place statically during compilation such that errors are caught at an early stage and do not leak into the end-product. Additionally, having such types available in a program allows a compiler to take advantage of this knowledge and perform optimisations. The disadvantage of static typing is that it imposes restrictions on the programs that can be defined in the programming language. An ideal static type system can decide for any program if it is well typed. However, there are properties for which there is no algorithm that can always decide if a program fulfills that property, such as termination as described by the halting problem. For such properties, a type system is always an approximation and therefore sometimes rejects programs for which the properties do hold. To refine this approximation, programming languages as well as their type systems are a constant subject of research.

1.1 Dynamic typing

The counterpart of static typing is dynamic typing, where verification takes place at run time. In dynamically typed languages it is mostly in the hands of the programmer to prevent erroneous behaviour. This is achieved by including guards and inspecting the types of values before coming to the actual operations, or simply by assuming that the program is correct. Consequently, any errors in a program will appear as late as when the program is executed. On the other hand, dynamic typing gives us the freedom to choose for ourselves if and when we look at the types of values, freeing us from the restrictions of static typing. Dynamic typing enlarges the number of programs that can be defined since we are not constrained by the approximation that a static type system imposes.

It might seem as there has to be made a distinct choice between static and dynamic typing in a programming language. However, having both in a single programming language is very well possible. Such a system gives us the best of both worlds: it allows us to detect errors at an early stage, as well as provides a way to circumvent the restrictions of a static type system.
The statically typed functional languages Clean and Haskell provide such a combination of static and dynamic typing, although there is a difference in their approach and expressivity. Clean offers dynamic typing in the language via a rich and mature built-in system, supporting both monomorphism (Abadi et al., 1991) and polymorphism (Leroy and Mauny, 1993; Abadi et al., 1995; Pil, 1997). Haskell provides dynamic typing via a library, one that is less expressive than the system of Clean since it only supports monomorphism (Baars and Swierstra, 2002; Cheney and Hinze, 2002). The advantage of having a built-in system is that the dynamic type system is defined on abstract syntax trees. These structures can be manipulated more freely in the implementation in the compiler, in contrast to a library which is restricted by the expressivity of the language itself. Also, a built-in system provides great flexibility in how dynamic typing is offered in the language syntactically. On the other hand, a library does not require any extension of the core language which reduces the complexity of the language and compiler.

Despite these differences their underlying philosophy is the same; this is best phrased by Meijer and Drayton (2004): “Static typing where possible, dynamic typing when needed”. The idea is that we start from a statically typed setting and provide a way out to perform dynamic typing. Of course, having such an escape to dynamic typing has its benefits but also comes at a price since erroneous behaviour can only be detected at run time. Hence, dynamic typing must be used with care, as put by Henglein (1992): “Only pay for the amount of dynamic typing that is unavoidable”. In essence, the use of dynamic typing is only really required when we deal with missing information at compile time. Typically this is the case when the types of values only become apparent at run time, for instance when we depend on a user who can provide us with any type of input. Another important reason is when we need to circumvent the static type system when it prevents us from defining a particular program.

The escape to dynamic typing is facilitated in Clean and Haskell in the same way. Both languages use a uniform black box that is used to store values that need to be dynamically typed. Such a dynamic value is itself of a special type that does not reveal the type of its contents. The value contained in a dynamic value is unwrapped at run time by pattern matching and comparing a specified type to the type that is stored together with the value. Of course, unwrapping a dynamic value can present an unexpected type, resulting in a run-time failure. Fortunately, the static type system guarantees that when pattern matching succeeds, the unwrapped value can be used in a type-safe fashion from there on. This approach facilitates type-safe transitions between the worlds of static and dynamic typing.

However, the interactions between the static and dynamic world are far from trivial. The corresponding type systems are separate in Clean and Haskell, not in the least because the nature of typing in these worlds is subtly different. Whereas the static type system verifies that a program is well typed, the dynamic type system merely compares a specified type against another type obtained from a dynamic value. These are different tasks with different resources at their disposal: static typing requires the complete program while dynamic typing can do so with types and descriptions of their definitions. Consequently, whenever the language is extended, the effects on the static and dynamic type system have to be considered separately.
1.2 Type-driven programming

There is a clear trend visible in functional languages such as Clean and Haskell that types are not just used to verify a program, but also play another role. A more type-driven style of programming is emerging where the behaviour of a program also depends on the types of the values involved.

Perhaps the earliest example of such type-driven programming is ad-hoc polymorphism via type classes (Wadler and Blott, 1989; Peterson and Jones, 1993). This form of polymorphism is a powerful abstraction mechanism to conveniently parameterise functions with behaviour. It allows us to give different meanings for a single function, so that the types occurring in the context of its application determine which meaning to use. Hence, types determine the behaviour of such an ad-hoc polymorphic function.

Another early form of type-driven programming is generic programming (Janssen and Jeuring, 1997; Backhouse et al., 1999; Hinze, 2000a; Alimarine and Plasmeijer, 2002). Many functions, such as testing for equality and printing of values, follow the same pattern. Generic programming reduces boiler-plate code in writing such functions over and over again by defining a function only once such that it can be applied to values of any type generically. This is achieved by defining a generic function on a universe that describes the structure of types, instead of the original values. Then, before we can deploy a generic function, we have to transform the original value to and from a value in the universe of the structure of types. Again, types play an essential role since their structure guides the behaviour of a generic function.

Not only can types determine the behaviour of functions, they can also determine the construction of values using datatypes. Generalised algebraic datatypes (Cheney and Hinze, 2003; Xi et al., 2003; Peyton Jones et al., 2006) are less liberal than algebraic datatypes since the type of a constructor is defined explicitly, thereby enforcing structural properties so to guarantee that the corresponding values are well formed. This even allows us to compute types from values, since we can compose constructors and thereby compose the attached types accordingly. Generalised algebraic datatypes really break the barrier that always existed between the construction of values and types.

More recently, the idea of type-driven programming has been taken to the next level where we also compute types from types, and not only define functions that are driven by types. Interestingly, type families (Chakravarty et al., 2005a & b; Schrijvers et al., 2008) allow us to also define functions at the type level. Although this brings new expressivity to the language, especially used in combination with generalised algebraic datatypes, such functions are limited compared to ordinary functions on values to ensure termination and completeness of the type system.

Finally, dependently-typed languages such as Agda (Norell, 2007) are a different breed of functional languages. In a sense, such languages take type-driven programming to an extreme since no real distinction is made between values and types: types are values. Consequently, functions can now take types as arguments as well as result in types, and types can even depend on values. However, the increased expressivity complicates the type system and commonly requires manual proofs from the programmer in exchange.
1.3 Outline of this thesis

All of the given examples of type-driven programming only concern static typing. However, dynamic typing is a form of type-driven programming as well. It involves making decisions based on the types of values, albeit at run time. This thesis describes in three parts how dynamic typing interacts with such static forms of type-driven programming.

The first part gives an overview of two functional languages, Clean and Haskell, that both play a leading role in this thesis. Chapter 2 describes the implementation of a double-edged front end for the Clean compiler to exchange sources between the two languages. This is achieved via two dialects of Clean and Haskell, dubbed Clean\(*\) and Haskell\(*\), that include just enough extra language features to use each other’s libraries conveniently. Haskell programs can now take advantage of Clean’s more expressive dynamic type system and are no longer limited by Haskell’s own less expressive library. This chapter also identifies the most salient differences between the two languages.

The second part describes the necessity of dynamic typing in type-driven programming. Chapter 3 discusses a Haskell library that makes extensive use of type-driven programming to generically rewrite terms using rewrite rules as values instead of functions. The library uses generalised algebraic datatypes to model heterogeneously typed metavariables and type families to generically extend datatypes with a constructor for metavariables. Dynamic typing turns out to be vital to define rewrite rules in terms of the original datatype instead of the inconvenient internal representations of the library. Chapter 4 focusses on the need for dynamic typing when manipulating heterogeneous structures described by generalised algebraic datatypes. Such structures often hide type information, complicating the definition of for instance a function that updates one of its values. This chapter shows that dynamic typing is needed to expose hidden type information, but that it is usually used in a cumbersome fashion. To make this more convenient, a new annotation is introduced and formally defined in terms of Clean’s dynamic typing.

The third part discusses how dynamic typing is accommodated in a language that facilitates type-driven programming. Chapter 5 describes the desire for a dynamic type system that is orthogonal to the static type system such that any value can be transferred between the two worlds. Meaning, there is no restriction on the values and types that can live in the dynamic world. As described earlier, every new aspect to static typing has an effect on the dynamic type system. Consequently, when a language facilitates type-driven programming in the static type system, the dynamic type system needs to move along to maintain the invariant of orthogonality. This chapter focusses on one of the earliest forms of type-driven programming, namely ad-hoc polymorphism via type classes. The design space of the interaction with Clean’s dynamic typing is explored, which brings a language where values seamlessly transfer between the static and dynamic world one step closer. Chapter 6 discusses what it takes for a functional language to embed polymorphic dynamic typing as a library, in contrast to Clean where this form of dynamic typing is built-in. Naturally, type-driven programming is of great importance here since value-level representations of types must be related to actual types to obtain a useful embedding. This chapter explores such an
embedding in Haskell using generalised algebraic datatypes and describes the difficulties with polymorphism. Being the ultimate form of type-driven programming, Agda provides the expressivity to perform the embedding of polymorphic dynamic typing and additionally prove its correctness using the language itself. Therewith, the parts are clarified where the additional power of a dependently-typed language is most needed, which gives great insight in what is required to perform such an embedding in Haskell.

In conclusion, this thesis shows that dynamic typing is an indispensable part of type-driven programming. It enriches the static forms of type-driven programming naturally since dynamic typing centers around types as well, providing an escape when static typing prevents the definition of a particular program. However, it is not straightforward to accommodate dynamic typing in a language that facilitates type-driven programming. Each and every one of its static forms has different and multiple interactions with dynamic typing. When embedding dynamic typing as a library, such as in Haskell, supporting all of these interactions quickly becomes complicated. Having a built-in system for dynamic typing, such as in Clean, seems to allow for better integration with the static forms of type-driven programming.

1.4 Origin of chapters

The chapters in this thesis originate from different papers and all have distinct contributions. The inevitable redundancy in introductory content is left unchanged so that the chapters can also be read independently. With each chapter, the origin, acknowledgements, and personal contributions are described:

- Chapter 2 is based on the Haskell ’10 paper *Exchanging sources between Clean and Haskell - A double-edged front end for the Clean compiler* (Van Groningen et al., 2010).

  I have initiated writing down the research and implementation performed by co-author John van Groningen. I have lead the writing process and contributed to every section of this chapter except for the section on performance, and mostly to the overall structure, the introduction, the sections on generic programming and dynamic typing, and the conclusion.

- Chapter 3 is based on the JFP ’10 paper *A lightweight approach to datatype-generic rewriting* (Van Noort et al., 2010c) and is an adaptation of an equally titled WGP ’08 paper (Van Noort et al., 2008). The present chapter includes several improvements over this previous work. Most prominently, while the library described in the earlier paper could only be used to generically rewrite values of regular datatypes, generic rewriting is now supported for a strictly larger class of datatypes, including types from families of mutually recursive datatypes. Furthermore, ill-formed rewrite rules are detected and guarded rewrite rules as well as heterogeneously typed metavariables are supported.

  The authors of the original paper would like to thank both Chris Eidhof and Sebastiaan Visser for their work on testing rewrite rules using generic test-data generation and Andres Löh for productive discussions on this work.
Finally, the authors are indebted to Doaitse Swierstra for his useful suggestions.

I was heavily involved in the research and implementation of the library described in the original paper, and have contributed to the renewed implementation initiated by co-author Stefan Holdermans. I have contributed to every section of this chapter except for the sections on the case study and performance, and written the section on detecting ill-formed rewrite rules.

• Chapter 4 is based on the IFL '09 paper *A typical synergy - Dynamic types and generalised algebraic datatypes* (Van Noort et al., 2010a).

I have independently performed and written down the research described in this chapter.

• Chapter 5 is based on the WGP '10 paper *Ad-hoc polymorphism and dynamic typing in a statically typed functional language* (Van Noort et al., 2010b).

The authors of the original paper are indebted to John van Groningen for the original idea of dictionary-passing types and the helpful discussions on the related subjects.

I have independently performed and written down the research described in this chapter.

• Chapter 6 is an adaptation of the WGP '11 paper *Embedding polymorphic dynamic typing* (Van Noort et al., 2011). The present chapter improves upon this previous work by using a simpler universe for the representation of polymorphic types and a less complex algorithm to determine if one type representation is an instance of another. More importantly, the approach described in the earlier paper includes several postulates, that have now been proven in Agda itself to hold, thereby making the cast function executable.

The authors of the original paper are indebted to James McKinna for invaluable discussions on the subject and pointing out the advantages of using environments over substitutions in the interpretation functions, and to Stefan Holdermans and Sjoerd Visscher for showing how to use type families to embed polymorphic dynamic typing in Haskell.

After first independently performing the research on the embedding of polymorphic dynamic typing in Haskell, I asked co-author Wouter Swierstra for his Agda expertise. Together, we performed the research on the embedding in a dependently-typed language. I have independently written down the research described in this chapter.
Part I

Functional languages
2 Exchanging sources between Clean and Haskell

Abstract

The functional languages Clean and Haskell have been around for over two decades. Over time, both languages have developed a large body of useful libraries and come with interesting language features. It is our primary goal to benefit from each other’s evolutionary results by facilitating the exchange of sources between Clean and Haskell and study the forthcoming interactions between their distinct languages features. This is achieved by using the existing Clean compiler as starting point, and implementing a double-edged front end for this compiler: it supports both standard Clean 2.1 and (currently a large part of) standard Haskell 98. Moreover, it allows both languages to seamlessly use many of each other’s language features that were alien to each other before. For instance, Haskell can now use uniqueness typing anywhere, and Clean can use newtypes efficiently. This has given birth to two new dialects of Clean and Haskell, dubbed Clean* and Haskell*.

Additionally, measurements of the performance of the new compiler indicate that it is on par with the flagship Haskell compiler GHC.

2.1 Introduction

The year of 1987 was a founding one for two pure, lazy, and statically typed functional languages. Clean (Brus et al., 1987) was presented to the public for the first time and the first steps towards a common functional language, later named Haskell, were taken (Hudak et al., 2007).

Clean was conceived at the Radboud University Nijmegen as a core language that is directly based on the computational model of functional term-graph rewriting to generate efficient code. It also serves as an intermediate language for the compilation of other functional languages (Koopman and Nöcker, 1988; Plasmeijer and Van Eekelen, 1993). For these reasons, it deliberately used a sparse syntax (Van Eekelen et al., 1990): “...at some points one can clearly recognize that [...] Clean is a compromise between a functional programming language and an intermediate language used to produce efficient code. For instance, a minimal amount of syntactic sugar is added in [...] Clean.”. Later, the core language was sugared. One particularly important factor was its adoption of uniqueness typing (Barendsen and Smetsers, 1993) to handle side-effects safely in a pure lazy language. Based on this concept, a GUI library (Achten et al., 1992; Achten and Plasmeijer, 1995) was developed, which was used in large applications such as
the Clean IDE, spreadsheet (De Hoon et al., 1995), and later the proof assistant Sparkle (De Mol et al., 2002). In 1994, Clean 1.0 appeared, which basically added the syntactic sugar to core Clean that was necessary to develop such large libraries and large applications. In the following years Clean turned open source, and extended its arsenal of functional language features with dynamic typing (Pil, 1999) and built-in generic programming (Alimarine and Plasmeijer, 2002), obtaining Clean 2.1 (Plasmeijer and Van Eekelen, 2002). Whenever we refer to Clean in this chapter, we mean this version.

Very shortly after the presentation of Clean, Haskell was born as a concepts language out of the minds of a large collaboration that idealised an open standard to “reduce unnecessary diversity in functional programming languages” and “be usable as a basis for further language research”. After three years, this effort resulted in the Haskell 1.0 standard (Hudak et al., 1992) and later the (revised) Haskell 98 standard (Peyton Jones and Hughes, 1999; Peyton Jones, 2003). Early this year, Haskell 2010 was announced and the Haskell’ standard is under current active development. Haskell especially enjoyed the benefits of a rapidly growing community; evolving and adapting standards quickly. The downside being that the term ‘Haskell’ became heavily overloaded. It is often not clear to what it refers: one of the standards, a specific implementation of the flagship Haskell compiler GHC, or something in between? Whenever we refer to Haskell in this chapter, we mean Haskell 98 and explicate any deviations.

Clean did not take part in the Haskell collaboration and chose to explore the world of functional programming on its own. After diverging onto different paths more than 20 years ago, we believe it is time to reap the benefits by exchanging (some of) each other’s evolutionary results. Both languages have developed interesting language features and concepts (e.g., uniqueness typing in Clean and monads with exceptions in Haskell) and many useful libraries (e.g., the workflow library iTask and the testing library Gast in Clean, and the parser combinator library Parsec and testing library QuickCheck in Haskell). Our long-term goal is to facilitate the exchange of such libraries and study the forthcoming interactions between languages features that are distinct to Clean or Haskell. There are many ways to achieve this goal. A naive approach is to define a new functional language that is the union of Clean and Haskell. The resulting language would become very baroque due to different syntax in Clean and Haskell for very similar, but not identical, concepts. A second approach is to develop two separate compilers that translate Clean to Haskell and vice versa. This would require an incredible amount of work and is quite hard since features from one language do not always easily project to the other language. This can be simplified by disallowing such features to be used in the libraries under exchange, but that restricts the application of libraries too much. Instead, we develop dialects of Clean and Haskell, dubbed Clean* and Haskell*, that include just enough extra language features to use each other’s libraries conveniently. Both new languages are realised in a double-edged front end for the Clean compiler that runs in two modes:

- Clean*, accepting Clean 2.1 programs extended with Haskell 98 features.
- Haskell*, accepting Haskell 98 programs extended with Clean 2.1 features.
Although Clean and Haskell are both pure and lazy functional languages, there are many subtle differences. An overview of most of the syntactic differences has been given in (Achten, 2007). In this chapter we mainly focus on the semantic differences and describe our effort to marry them within the two extended languages. We do not aim to give a complete and detailed overview, but instead identify the biggest challenges and describe the intuition behind their solution and implementation. Concretely, our contributions are the following:

- We identify the most salient differences between Clean and Haskell: modules, functions, macros, newtypes, type classes, uniqueness typing, monads, records, arrays, dynamic typing, and generic functions (Sections 2.2 to 2.12).

- With each difference we discuss if and how Clean* and Haskell* support the exchange and briefly explain how this is incorporated in an implementation.

- We provide a concrete implementation of the front end that supports Clean, Haskell, and their dialects Clean* and Haskell*.

We give a brief comparison of the current performance of the front end in relation to GHC (Section 2.13). We end this chapter with related work (Section 2.14) and conclude with a discussion and future work (Section 2.15).

Since Clean and Haskell are syntactically so much alike, it can be quite hard to disambiguate examples from both languages. Therefore, we choose to start each code fragment with a comment line, // Clean or -- Haskell respectively, choosing redundancy over opacity. Similarly for the dialects of the languages, we start with a comment line // Clean* or -- Haskell*. Also, since this chapter is very much concerned with the exact concrete syntax, the examples given are typeset in verbatim.

## 2.2 Modules

Clean and Haskell come with many libraries. Instead of migrating these manually, we aim to support the exchange of sources via the front end. It allows Clean modules to import Haskell modules and vice versa. In this section we first briefly compare the two module systems (Section 2.2.1) and corresponding compilation strategies (Section 2.2.2). Then we discuss how the front end facilitates mixed compilation of modules in Clean* and Haskell* (Section 2.2.3).

### 2.2.1 Module systems

From the beginning, Clean has used a module system that is very similar to that of Modula-2 (Wirth, 1985). Implementation modules reside in .icl files and contain all implementations of functions, datastructures, and type classes. Definition modules reside in .dcl files and specify the corresponding interfaces by the exported definitions. Besides importing an entire module, Clean allows the explicit import

---

1The front end is under active development, current releases are available on the Clean Wiki via [http://wiki.clean.cs.ru.nl/](http://wiki.clean.cs.ru.nl/).
of elements of a module, distinguishing between the sort of element (functions, types, type classes, etc.). This has been included in Haskell* during this project.

Although Haskell 1.0 also used a module system with separate module interfaces, these were abandoned as of Haskell 1.3 because they were increasingly perceived as compiler-generated artifacts, rather than interface definitions (Hudak et al., 2007). Instead, the header of a module enumerates its exported symbols. This perception fits within the language philosophy of Haskell to have the programmer specify only what is required to successfully compile a program. For instance, in Haskell it is allowed to export an identifier $x$ in a module $M$ but not its type, and to import $x$ in another module $N$. Because the type of $x$ is not in scope in module $N$, it cannot be given an explicit type. However, the compiler can, and has to, find this type by inspecting module $M$. Haskell prescribes no relation between module names and files, but by convention each module resides in a .hs or .lhs file. Haskell provides fine-grained control over the names of imported definitions. This is achieved via hiding specific definitions, qualified imports of modules, and hierarchical modules (this last feature is an extension of Haskell). These constructs have been included in Clean* during this project.

User-defined definition modules as used in Clean have as advantage that a programmer obtains a clear description of the offered interface of a specific library module, which is very useful from an engineering point of view. A disadvantage of the approach is that a definition module cannot be used by a compiler to provide additional information about the actual implementation, which might be used for optimisations such as inlining.

### 2.2.2 Compilation strategies

When the Clean compiler compiles an implementation module, it is first verified that the exported definitions match the corresponding implementation. Imported definition modules are assumed to match their implementation and an implementation module is only recompiled if it is new, or when required by its timestamp. Compilation of modules takes place from top to bottom. When the compiled version of an imported module is up to date, it suffices to inspect only the definition modules of the imported modules, which significantly speeds up the compilation process. Clean modules are compiled to intermediate ABC code (Koopman et al., 1995), from which object code is generated.

The compilation process of a Haskell program is more involved. Because modules can confine themselves to exporting definitions only, but not their types, all sources of imported modules must be available. During compilation, interface files are generated that can be used instead. In the end, object files are generated that are used by a linker to create an executable.

### 2.2.3 Mixed compilation

The support of mixing Clean* or Haskell* modules in the Clean compiler is based on definition modules. In the Clean world, these definition modules are still defined separately. The definition module of a Haskell* module is generated by the compiler. When Clean* and Haskell* modules are mixed, the compiler has to switch
between compilation strategies: Clean* modules are compiled top down as usual, while Haskell* modules have to be compiled bottom up in order to generate the required definition modules. The compiler has to know with what kind of module it is dealing with. If the module is a .icl file, it is assumed that there is a manually defined .dcl file available. Otherwise, if the module is an .hs or .lhs file, an accompanying .dcl file is generated. If a previous compilation of a Haskell* module already generated such a definition module, the new definition module is compared to the old one. If they are identical, the old definition module is kept, leaving its timestamp unchanged. Otherwise, it is replaced by the new definition module. Before a module is compiled, the definition modules of all imported modules have to be available. If these do not exist or are out of date, since their timestamp is newer than the one of the definition module, the corresponding Haskell* modules have to be compiled first in order to generate the required definition modules. As we will see in the following sections, generated definition modules from Haskell* modules sometimes include additional information to inform the compiler of typical Haskell* constructs. For efficiency reasons it is sometimes worthwhile to define definition modules of Haskell modules by hand. In Section 2.6 we see an example where we manually include specialisation information in exported function types.

2.3 Functions

The semantics of the core of Clean is based on term-graph rewriting. The expression that is computed is a computation graph and functions are sugared versions of term-graph rewrite rules. In Clean, the signature of a function reveals information about its arity, strictness, and uniqueness properties. The first two concepts are discussed in this section, the third in Section 2.7.

Sharing is explicit in Clean functions. Variable names in function argument patterns, and case patterns as well, really point to a subgraph in the computation graph after matching a redex. Having multiple occurrences of these variables on the right-hand side of a function and case patterns implies that these are shared. Similarly, constants defined locally using let or where, also called local graph definitions, on the right-hand side of a function are also always shared. Local function definitions are not shared and are always lambda lifted. In all cases, = is used as a separator between the left-hand side and right-hand side of a function or local definition. If the programmer intends to locally define a constant but with the nonsharing behaviour of a function, this is denoted using => as a separator, or by providing an explicit type signature. Haskell does not explicitly specify what must be shared, but every implementation uses similar rules as stated above. At the top level of a Clean module, every definition is considered to be a function definition. If the programmer intends a constant in applicative form (CAF), this is denoted by using =: as a separator. As an example, we define the well-known efficient list of Fibonacci numbers as a constant:

// Clean
fibs =: [1 : 1 : [x + y \ x <- fibs & y <- tl fibs]]

If we used = as a separator instead, this would result in recomputing the list for each invocation.
In Haskell, a top-level function without arguments is assumed to be a CAF, unless it has an explicit overloaded type signature. Hence, the above example can be expressed as follows without risk of recomputation:

```haskell
-- Haskell
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

In Clean the programmer can make the tradeoff between (possible) recomputation and space usage. In Haskell this choice is fixed to storing the results and hence usage of space.

The arity of term-graph rewrite rules can be greater than one, in contrast to functions considered from a \(\lambda\)-calculus perspective as in Haskell. For this reason, function signatures in Clean show the arity of their implementation, while signatures are curried in Haskell. The advantage to knowing the arity of a function is efficiency: a function application knows when it is fully saturated. It is important to observe that this is a syntactic issue: it neither limits the type system nor the use of currying in Clean. As an example, consider the following function that combines the application of the well-known functions `map` and `concat` (named `flatten` in Clean):

```clean
// Clean
concatMap :: (a -> [b]) [a] -> [b]
concatMap f xs = flatten (map f xs)
```

The function type exposes the arity of the implementation, which is two in this case. Hence, if we change the definition to a point-free notation, the type of the function changes. We use the infix Clean function \(\circ\) for function composition, in contrast to Haskell’s Prelude dot-notation:

```clean
// Clean
concatMap :: (a -> [b]) -> ([a] -> [b])
concatMap f = flatten \(\circ\) map f
```

It should be noted that, as usual, the right-most brackets can be omitted because \(\rightarrow\) associates to the right. Now, the arity of the function is one, which is reflected in its type by the insertion of a function type. Moving the first argument inwards changes the arity of the type again, making it of arity zero:

```clean
// Clean
concatMap :: ((a -> [b]) -> [a] -> [b])
concatMap = \f -> flatten \(\circ\) map f
```

In Haskell, all these implementations are given the same type, namely:

```haskell
-- Haskell
concatMap :: (a -> [b]) -> [a] -> [b]
```

Consequently, such a type does not reflect the arity of its implementation.

Similar effects occur in the use of type synonyms in function signatures. Suppose that we define the following type synonym:
2.3 Functions

// Clean
:: ListF a b == a -> [b]

-- Haskell
type ListF a b = a -> [b]

In Haskell, ListF a b -> ListF [a] b is also a valid type for any of the implementations of concatMap, but in Clean (ListF a b) -> ListF [a] b is only valid for the second definition with arity one.

Since its first version, Clean comes with a strictness analyser (Nöcker, 1994) as well as strictness annotations for function signatures. Strictness information is crucial for generating efficient code. The programmer can add strictness annotations to function arguments, and hence export this information in the corresponding definition module. Haskell has no support for strictness information in function signatures. Clean and Haskell both support strictness annotations in datatypes in very similar ways, therefore this is not discussed.

Exchange

Clean* functions can be used easily by Haskell* and vice versa without modification. Haskell* function definitions are interpreted as term-graph rewrite rules as described above. In Haskell* function signatures can be given strictness annotations in the same fashion as in Clean*. Strictness information is derived during compilation and exported in the corresponding definition module. Below is discussed how the arity information is derived and exported.

Implementation

The issue with function arity shows up in interfaces between Clean* and Haskell* modules. The front end transforms user-provided Haskell* types for exported functions in the generated definition module and makes the arity of a Haskell* function explicit. Suppose we have the following Haskell* definition of the concatMap function:

-- Haskell*
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f xs = concat (map f xs)

When a Haskell* module exports this function, the front end generates a Clean type for the definition module that reflects the arity of the implementation, which is two in this case:

concatMap :: (a -> [b]) [a] -> [b]

If we define this function in point-free notation, the arity of the implementation changes and the exported type becomes:

concatMap :: (a -> [b]) -> [a] -> [b]
Note that in this case, the exported type is syntactically identical to the original Haskell type, but explicitly states that `concatMap f` yields a function value.

Similarly, when a type synonym obscures the arity of a function, its exported type is transformed. Suppose we export the following functions with one identical Haskell type:

```haskell
-- Haskell
concatMap2, concatMap1, concatMap0 :: ListF a b -> ListF [a] b
concatMap2 f xs = concat (map f xs)
concatMap1 f = \xs -> concat (map f xs)
concatMap0 = \f xs -> concat (map f xs)
```

With each version, the type synonym is expanded to match the arity of the implementation of the function. Thus, the definition module contains:

```clean
concatMap2 :: (a -> [b]) ![a] -> [b]
concatMap1 :: (a -> [b]) -> [a] -> [b]
concatMap0 :: ((a -> [b]) -> [a] -> [b])
```

Only `concatMap2` is strict in its list argument since `concat` and `map` are strict, and the other definitions return functions that still expect one or two arguments.

## 2.4 Macros

Clean 0.8 added macros to the language. A macro can be regarded as a function with one alternative and just named arguments. Macros are substituted at compile time, and hence are not allowed to be recursive. Naturally, it may use other recursive functions or define recursive functions locally. Note that the substitution is a graph reduction, and not a textual substitution. For instance, we define a macro to double a value:

```clean
// Clean
double x ::= x + x
```

The application `double (fibs !! 100)` is reduced at compile time to the expression `let x = fibs !! 100 in x + x`. Hence, the computation of `x` is shared.

In Haskell, the programmer can use the `INLINE` pragma to encourage the compiler to inline the body of a function. For instance, the above macro is defined as follows in Haskell as a function to be inlined:

```haskell
{-# INLINE double #-}
double x = x + x
```

**Exchange**

Haskell* modules can import and use Clean macros, and define them using the same syntax. The `INLINE` pragma is not yet included in Clean*. However, macros subsume this concept.
Implementation

Currently, it remains future work to export macros from Haskell*.

2.5 Newtypes

Although type synonyms are useful to document code and explain the purpose of a type, they suffer from the disadvantage that they cannot serve as an instance of a type class or be recursive. Clean’s syntax for type synonyms indicates that they are just macros at the type level. Haskell 1.3 introduces newtype declarations (i.e., datatype renamings) which are syntactically identical to an algebraic datatype with exactly one constructor of arity one, but with the intention to behave semantically as a type synonym. For instance, here are two newtype definitions:

```haskell
newtype Nat = Nat Int
newtype Fix f = In (f (Fix f))
```

This eliminates the above mentioned drawbacks: Nat can be made an instance of say the type class Integral, and Fix is clearly a recursive type. The constructors are still included in patterns and construction, but are assumed to be erased by the compiler. Hence, every Nat instance behaves as an ordinary Int value and every Fix f behaves as a plain recursive value.

Clean does not support newtypes. The best approximation is to use an algebraic datatype with a strict argument:

```clean
:: Nat = Nat !Int
:: Fix f = In !(f (Fix f))
```

Operationally, this version is more expensive than a version where these constructors are erased at compile time.

Exchange

All Haskell* newtypes can be imported and used in Clean* modules and adhere to the assumed Haskell semantics. The mentioned Clean types are defined as newtypes in Clean* as follows:

```clean*
:: Nat =: Nat Int
:: Fix f =: In (f (Fix f))
```

Note that this code fragment is also legal Haskell*.

Implementation

The implementation of newtypes avoids the constructor overhead since all constructors belonging to newtypes are erased at compile time. Removing constructors is not as trivial as it seems. For example, consider the following Haskell wrapper function toNat:
2 Exchanging sources between Clean and Haskell

```haskell
-- Haskell
toNat :: Int -> Nat
toNat = Nat
```

We have to introduce an identity function if the constructor \texttt{Nat} is erased. Also, constructors need to be erased from patterns in function definitions:

```haskell
-- Haskell
fromNat :: Nat -> Int
fromNat (Nat _) = 10
```

If we would leave the constructor, the function becomes strict while the semantics requires a nonstrict function. The value \texttt{fromNat \bot} evaluates to the value 10 and not to \bot.

Also, the newtypes itself are erased at compile time. This implies that \texttt{Nat} has to be replaced by \texttt{Int} in the above examples. Evidently, erasure is more intricate when recursive newtypes are involved. Newtypes must also be erased in order to make annotations for uniqueness typing on the argument of the newtype effective. The type wrapped in the newtype obtains the type annotations of the newtype definition.

### 2.6 Type classes

Haskell has supported type classes from the very beginning. Clean, having started as a core language, added type classes to the language with version 1.0 in 1994. There are a number of differences that need to be discussed.

While Clean supports multi-parameter type classes, the parameters of a Haskell type class are restricted to one (although many Haskell implementations allow more parameters). For example, consider the following type class \texttt{Array \ a \ e} that is used for arrays of type \ a \ with elements of type \ e, as we will see in Section 2.10:

```clean
// Clean
class Array a e where
    createArray :: Int e -> (a e)
    size :: (a e) -> Int
```

Type classes in Haskell can suggest default implementations for its members that can be overruled in specific instances. For instance in the equality type class:

```haskell
-- Haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

    x == y = not (x /= y)
    x /= y = not (x == y)
```

If an instance provides no definition, the default definition is used. In Clean, default members are defined using macros, which are described earlier in Section 2.4:
2.6 Type classes

// Clean
class Eq a where
  (==) :: a a -> Bool

  (/=) x y ::= not (x == y)

The difference with Haskell is that default members via macros cannot be re-defined.

In contrast to Haskell, Clean does support defaults on the level of instances. For example, consider the catch-all instance for Eq:

// Clean
instance Eq a where
  _ == _ = False

This instance is used whenever no other instance matches. Consequently, overlap can occur between instances, but this is only allowed with such a catch-all instance. We cannot define both instances of Eq for both (Int, a) and (a, Int) in Clean.

As we discussed in Section 2.3, Clean enforces an explicit arity of function type signatures while Haskell types do not reflect the arity of their implementation. Hence, the members of the instances of a Clean type class must agree on their arity as specified by the type class. Instances of a Haskell type class can differ in arity from each other and the original type class definition.

To avoid the overhead of the dictionary-passing style translation of type classes, Haskell includes the SPECIALIZE language pragma to generate specialised versions at compile time. For instance, in the overloaded equality on lists, we indicate that specialised definitions for Int and Bool are to be generated and used wherever possible:

-- Haskell
{-# SPECIALIZE eqL :: [Int] -> [Int] -> Bool #-}
{-# SPECIALIZE eqL :: [Bool] -> [Bool] -> Bool #-}

eqL :: Eq a => [a] -> [a] -> Bool

eqL [] [] = True

eqL [] _ = False

eqL _ [] = False

eqL (x:xs) (y:ys) = x == y && eqL xs ys

In Clean, any overloaded function is specialised within module boundaries. Therefore, only exported functions and instances possibly need to be specialised using the keyword special in a definition module:

// Clean
eqL :: [a] [a] -> Bool | Eq a special a = Int; a = Bool

instance Eq [a] | Eq a special a = Int; a = Bool

In contrast to Haskell, such specialisations are specified by a substitution of type variables instead of a substituted type.
To avoid boilerplate programming, Haskell supports a **deriving** clause for **data** and **newtype** declarations. This relieves the programmer from writing instances of the type classes **Eq**, **Ord**, **Enum**, **Bounded**, **Show**, **Read**, and **Ix** herself, but instead lets the compiler do the job. In Clean, this kind of type-driven programming is achieved using generic functions, as we will discuss later in Section 2.12.

Haskell uses a rather elaborate system of type classes to organise numerical values: **Num**, **Real**, **Fractional**, **Integral**, **RealFrac**, **Floating**, and **RealFloat** for handling values of type **Int**, **Integer**, **Float**, **Double**, and **Rational**. Numeric denotations are overloaded: 0 is of the type **Num a => a** and is in fact the expression `fromInteger (0 :: Integer)`. Therefore, a Haskell programmer needs to add a type signature to disambiguate overloading from time to time. A **default** declaration provides another approach to disambiguate these cases. This consists of a sequence of types that are instances of the numeric type classes. In case of an ambiguous overloaded type variable that uses at least one numeric type class, the sequence of types is tried in order to find the first instance that satisfies the constraints. A module has at most one such declaration, and by default it is **default (Integer, Double)**. Clean uses a much simpler approach: numbers are either integer, **Int**, or floating point, **Real**, and their denotations are different: 0 is always of type **Int**, and 0.0 is always of type **Real**. Coercion between these types is achieved explicitly using any of the overloaded functions **toInt**, **toReal**, **fromInt**, or **fromReal**.

**Exchange**

Haskell* supports the less restrictive multi-parameter type classes of Clean. Not only can we import such definitions in Haskell*, we can also define such type classes ourselves and provide instances.

When importing a type class from the other language, the semantics of default members remains the same: Clean* can redefine Haskell default members while Haskell* cannot redefine Clean macros.

The arity of the members of a concrete instance is determined by the importing language. Members of an instance of a Clean type class in Haskell* can be of any arity, while the arity of the members of a Haskell type class in Clean* is the number of arguments.

Specialisation in the style of Haskell is not yet implemented. Recall that specialised definitions are generated within module boundaries, similar to Clean.

The type class hierarchy for numerical values in Haskell is available in Clean* as a library. Clean’s types for numerical values are currently not supported in Clean*. However, Haskell* can use Clean’s numerical types by prefixing such a value with ‘. The value ‘0’ is of the Clean type **Int**, just like the Haskell value 0 :: **Int**. Similarly, the value ‘0.0’ is of the Clean type **Real** like the Haskell value 0.0 :: **Double**. Proper support for efficient **Float** values in Haskell* is still under active development.

**Implementation**

The front end uses Clean macros to implement default members in Haskell*. The default members can be redefined, but their current form is restricted. A default
member in Haskell\* must have the same arity as the type it has been given, it can only consist of one alternative, and no infix-style definition is allowed. Also, such default members cannot be exported yet, this is future work.

Since the arity of members of Haskell instances can differ, the generated definition module of a Haskell\* module must include the types of the exported instance members to reflect their arity.

To facilitate efficient implementations of some of the Haskell Prelude functions, Clean includes redefinitions of exported specialised instances and functions. Consider the following exported Haskell function that converts `Integral` values:

```haskell
fromIntegral :: (Integral a, Num b) => a -> b special a = Int, b = Double
            ::= fromIntegralIntDouble
```

Here, we manually include a type signature in the definition module that defers the specialisation to a more efficient implementation in `fromIntegralIntDouble`.

Derived instances in Haskell\* are automatically included in the generated definition module such that these can be imported from another module. The implementation of the deriving construct in Haskell\* is not as straightforward as it may seem. If some of the derived instances are already defined but themselves have a more complicated context, a fixed-point computation is required to determine the context by reduction.

In Clean, CAFs are not allowed to be overloaded since such a value must have a single type in order to be a proper constant. In Haskell, overloaded CAFs without an explicit type signature are allowed, but overloading is resolved at compile time using the monomorphism restriction and the default rule as described earlier. Consequently, the type of an overloaded CAF cannot be determined just using its definition and the types of the functions that it uses, but also requires all the uses of the CAF in the module. Therefore, we may have to type check the entire module before we can determine the type of the CAF. The following implementation is used:

1. The type of a CAF \( c \) is determined without the monomorphism restriction and default rule. If \( c \) is not overloaded, type checking continues as usual.

2. If the CAF \( c \) is overloaded and used by another function \( f \), a preliminary type of \( f \) is determined using the overloaded type of \( c \). The type of the use of \( c \), after unification, is remembered. If \( f \) contains more than one use of \( c \), the types of all uses are unified. Other CAFs that are used are also remembered together with their types.

3. If the function \( f \) with a preliminary type is used by another function \( g \), then \( g \) is typed as if \( g \) uses the CAFs remembered in the preliminary type. Hence, a preliminary type is inferred for \( g \) that contains the types of the CAFs that are (indirectly) used. CAFs that use other CAFs are treated similarly.

4. The remembered preliminary types of the CAFs are unified to determine their types.

5. All functions for which preliminary types were inferred are type checked again, but now using the no longer overloaded types of the CAFs.


2.7 Uniqueness typing

Uniqueness typing relies heavily on the fact that sharing is completely explicit in Clean, as discussed in Section 2.3. A value that is unique has a single path from the root of the computation graph to the value. A function demands such an argument using the * annotation in its signature. Function bodies that violate this constraint are not well typed, and hence are rejected during compilation. Values that have a single reference can be updated destructively without compromising referential transparency. This allows Clean to support arrays with in-place updates of its elements, as we discuss later in Section 2.10. The programmer can annotate function arguments and datatypes with uniqueness attributes for the same purpose. Uniqueness can also be used to implement I/O, by annotating values that are somehow connected with the ‘outside’ world as being unique, which is discussed in Section 2.8.

As an example of uniqueness typing, consider a stateful map function, mapS, that threads a unique state of type *s. Note that type variables need to be attributed uniformly:

// Clean
mapS :: (a *s -> (b, *s)) [a] *s -> ([b], *s)
mapS _ [] s = ([], s)
mapS f [x:xs] s = ([y:ys], s2)
  where
    (y, s1) = f x s
    (ys, s2) = mapS f xs s1

Actually, the most general type for mapS is one that allows both nonunique and unique arguments. The . annotation ensures that the same type variable is assigned the same uniqueness attribute:

// Clean
mapS :: (.a .s -> (.b, .s)) [.a] .s -> ([.b], .s)

The type variable .a is either unique or nonunique in the signature, the same holds for .b and .s. For reasons of presentation, we usually omit these extensive type signatures.

The world-as-value programming style is supported syntactically in Clean using #-definitions, also known as let-before definitions. For instance, mapS is preferably written as:

// Clean
mapS :: (.a .s -> (.b, .s)) [.a] .s -> ([.b], .s)
mapS _ [] s = ([], s)
mapS f [x:xs] s # (y, s) = f x s
  # (ys, s) = mapS f xs s
  = ([y:ys], s)

Note that this definition is a sugared version of the earlier mapS definition using local where definitions.
Exchange

Haskell\* accepts uniqueness typing in Clean style. It can use Clean functions that manipulate unique values. As an example, here is a function that uses Clean I/O to write data to a file using an accumulating parameter:

```haskell
-- Haskell*
writeLines :: Show a => [a] -> *File -> *File
writeLines [] file = file
writeLines (x:xs) file =
    writeLines xs (fwrites (clstring (show x)) file)
```

We use Clean's StdFile library function `fwrites` to write a string to a file and `clstring` to convert a Haskell string to a Clean string (their difference is discussed in Section 2.10).

Naturally, the uniqueness properties of Haskell\* functions need to be verified. Types can be annotated with uniqueness attributes explicitly, or uniqueness information is derived and exported in the corresponding generated definition module. For instance, consider this Haskell\* function to update an element in a list:

```haskell
-- Haskell*
updateAt _ _ [] = []
updateAt 0 x (_:ys) = x \ y
updateAt n x (y:ys) = y : updateAt (n - 1) x ys
```

This function can be applied to a list that may contain unique values (.a) and preserves the uniqueness \( u \) of the spine of the list (u:[.a]):

```haskell
-- Haskell*
updateAt :: Num n => n -> .a -> u:[.a] -> u:[.a]
```

The uniqueness attributes in this type are identical to those of `updateAt` in Clean's StdList module.

Uniqueness annotations can also enforce constraints. Consider the following function to swap an element in a possibly spine-unique list, instead of updating it:

```haskell
-- Haskell*
swapAt :: Int -> .b -> u:.b] -> (u:.b], v:.b]), [u <= v]
swapAt _ x [] = (x, [])
swapAt 0 x (y:ys) = (y, x:ys)
swapAt n x (y:ys) = (z, y:zs)
    where
        (z, zs) = swapAt (n - 1) x ys
```

The source and result list now have different uniqueness attributes (\( u \) and \( v \) respectively), but they are related in the sense that the uniqueness of the source list is at least as unique as the result list (\([u <= v]\)). In this case it means that from a nonunique source list you cannot construct a unique result list (due to sharing of part of the list spine), but from a unique source list you can construct a nonunique or a unique result list.
Implementation

The issues that are related to the monomorphism restriction and default rule, as discussed earlier in Section 2.6, are solved in order to adopt Clean’s uniqueness typing in Haskell*.

2.8 Monads

Any practical programming language needs to be able to describe interactions with the ‘outside’ world. Clean and Haskell have followed entirely different solutions for this challenge. In Clean 0.8, uniqueness typing has been included to support an explicit environment-passing style (i.e., the world-as-value style). In Haskell 1.3, monads were adopted in favour of the stream-based and continuation-based I/O of earlier Haskell versions.

The basic philosophy of monads is that a monadic value represents a recipe that, when performed, may have side-effects and yields a value of some type. A monad consists of implementations for the combinators return and >>=:

```haskell
-- Haskell
infixl 1 >>=
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

A well-known instance of this type class passes a state of type \( s \) from function to function. The state-passing function is wrapped in the newtype \( \text{StateF} \):

```haskell
-- Haskell
newtype StateF s b = StateF (s -> (b, s))

instance Monad (StateF s) where
  return x = StateF (\s -> (x, s))
  (StateF f) >>= g = StateF (\s -> let (x, s1) = f s
                                 in g x)
```  

A very similar type class \texttt{Monad} is defined in Clean:

```clean
// Clean
class Monad m where
  return :: a -> m a
  (>>=) infixl 1 :: (m a) (a -> m b) -> m b

The differences with the Haskell definition are the notation for the fixity of the >>= combinator and the explicit arity in the types.

Instead of a newtype for \texttt{StateF} we use an algebraic datatype, as described in Section 2.5. It should be noted that additional uniqueness attributes are required in the right-hand side of \texttt{StateF} to allow both \( b \) and \( s \) to be unique. We rely on uniqueness typing to ensure a correct single-threaded implementation:
Monads are used to structure programs. Using this Monad type class, the function mapS from Section 2.7 is expressed more elegantly:

```
// Clean
mapS :: (a -> m b) [a] -> m [b] | Monad m
mapS _ [] = return []
mapS f [x:xs] = f x >>= \y ->
    mapS f xs >>= \ys ->
    return [y:ys]
```

In Haskell, monads are supported syntactically with do-notation. Hence we can choose for the definition of mapS for a notation similar to the Clean version or the version with do-notation:

```
-- Haskell
mapS :: Monad m => (a -> m b) -> [a] -> m [b]
mapS _ [] = return []
mapS f (x:xs) = do y <- f x
    ys <- mapS f xs
    return (y:ys)
```

The IO monad in Haskell is used to sequence I/O operations. The world is hidden from the programmer, and hence there is no danger of violating the single threadedness of this value. In Clean, the world is not hidden from the programmer, and single threadedness is guaranteed by marking them unique. The programmer either chooses to pass these objects explicitly as in the previous section, or to hide the unique object in a monad and pass it implicitly.

The IO monad in Haskell also enables exception handling. Its single thread-ness ensures a correct binding of exceptions to handlers in a lazy language.

**Exchange**

Monads are integrated seamlessly with uniqueness typing. In the previous section we explained that uniqueness typing is available in Haskell*. The IO monad, as well as conversions from and to a unique world, is available in Clean* via:

```
-- Haskell*
newtype IO a = IO (!*World -> *(a, !*World))
```

Since this is an ordinary type, it is straightforward to pack a unique world in an IO value and to unpack it again.
Implementation

The basic transformation scheme from do-notation to ordinary monadic constructors is given by Peyton Jones (2003). In order to achieve efficient execution, the code obtained by this transformation needs to be optimised. Currently our implementation of Clean performs a number of optimisations, such as inlining the member definitions of the IO instance for Monad.

Also, the exception-handling mechanism is implemented in both Clean and Haskell. The implementation maintains a stack of exception handlers and dynamically searches for the correct handler if an exception occurs. This makes installation of a handler via a catch relatively expensive, but prevents costs during ordinary evaluation.

2.9 Records

Records were introduced in Clean 1.0. A Clean record is an algebraic datatype of one alternative that does not have a constructor, but a nonempty set of field names. Records are allowed to use the same (sub)set of field names. For instance, the following declarations happily coexist:

```clean
// Clean
:: GenTree a = {elt :: a, kids :: [GenTree a]}
:: Stream a = {elt :: a, str :: Stream a}
```

Field values are extracted via pattern matching on the field names or by using a field name as a selector. In case of overlapping field names, a programmer must disambiguate the expression by either providing one distinguishing field name in a pattern (e.g., \{elt, kids\} and \{elt, str\}) or by inserting the appropriate type constructor name (e.g., \{GenTree | elt\} in a pattern or \x.GenTree.elt as a selector).

Records are created by exhaustively enumerating all field names or by updating a subset of the field names of an existing record. Here is an example of a function that updates an element of a stream:

```clean
// Clean
updStream :: Int a (Stream a) -> Stream a
updStream i x s =: {str}
  | i < 0 =: s
  | i == 0 =: {Stream | s & elt = x}
  | otherwise =: {s & str = updStream (i - 1) x str}
```

Haskell supports records only partially (since Haskell 1.3) in the form of field labels. All arguments of a constructor of an algebraic datatype are either addressed by their position or by field labels. A field label \(f\) is allowed in several alternatives of an algebraic datatype \(T\), provided they have the same type \(a\). Every field label brings a new function in scope, named \(f :: T \rightarrow a\). For this reason, no two datatypes can use the same field label, even if they have the same result type.

To create a record, the corresponding constructor must be provided and a (possibly empty) set of field labels to be initialised. Any omitted nonstrict field label
is silently initialised by \( \bot \). It is illegal to omit strict field labels at initialisation. Given a record value, a new record is created by updating a subset of the field labels. As an example, the `Stream` datatype and the `updStream` function look as follows in Haskell:

```haskell
-- Haskell
data Stream a = Stream {elt :: a, str :: Stream a}

updStream :: Int -> a -> Stream a -> Stream a
updStream i x s@(Stream {str = str})
  | i < 0 = s
  | i == 0 = s {elt = x}
  | otherwise = s {str = updStream (i - 1) x str}
```

**Exchange**

We allow both styles of records: a Clean* program can still define record types with overlapping field names, and a Haskell* program can define record types with multiple alternatives that use the same field labels. In Haskell*, it is allowed to import and use Clean records. Clean record fields are selected with ~, and the record type can be used to disambiguate field names. For instance, the Clean `GenTree` and `Stream` record types can be imported and used in the same Haskell* module:

```haskell*
-- Haskell*
mkGenTree :: GenTree a
mkGenTree = {elt = 0, kids = []}

mkStream :: Stream a
mkStream = {elt = 0, str = mkStream}

rootGenTree :: GenTree a -> a
rootGenTree t = t~GenTree~elt
```

The `mkGenTree` and `mkStream` functions construct a Clean record.

Conversely, a Clean* module can import Haskell records and their field selector functions as well. For instance, a Haskell module that exports the above definition of `Stream` can be used in Clean*:

```// Clean*
mkStream :: Stream a
mkStream = Stream 0 mkStream

hdStream :: (Stream a) -> a
hdStream s = elt s
```

A Haskell record is defined as a vanilla algebraic datatype. Clean* does not support the field label syntax at Haskell record value construction.
Implementation

The mixed use of Clean records in Haskell gives rise to several parser issues. Consider the following example:

```haskell
-- Haskell
analyseThis = C {elt = 0, kids = []}
```

This is either a normal Haskell record update where `C` has the type `GenTree a`, or the function `C` applied to a Clean record, but also a data constructor `C` with a Clean record of type `Stream a`:

```haskell
-- Haskell
data T a = C (Stream a)
```

In Haskell, the programmer can switch between layout-sensitive and layout-insensitive definitions within a function body. Layout-sensitive mode is assumed when no opening brace is encountered after one of the keywords `where`, `let`, `do`, or `of`. In Clean, layout-insensitive mode is switched on or off at the beginning of an entire module, simply by ending the module header with `;` (on) or not (off). Hence in Haskell, using a local definition that pattern-matches a Clean record is very similar to a local layout-insensitive definition. Consider the two following definitions:

```haskell
-- Haskell
f = (elt, kids) where {elt = 3; kids = []}
g = (e, k) where {elt = e, kids = [k]} = mkGenTree
```

Here, it can only be determined that a local layout-insensitive definition is given due to the use of `;` and missing `= ...` right-hand side. Currently, Haskell allows switching to layout-insensitive mode via `{`, but does not allow switching back.

### 2.10 Arrays

Clean has extensive language support for efficient arrays that can be updated destructively due to their uniqueness properties. Arrays with elements of type `a` come in three flavours: lazy (`{a}`), which is the default), strict (`{!a}`), and unboxed (`{#a}`). Since these are different types, array operations are organised as a multi-parameter type class `Array a e` where `a` is the array type, and `e` the element type. Array operations are bundled in the module `StdArray`. Unboxed array elements can only be basic types, arrays, or records. Note that in Clean the `String` type is implemented as an unboxed array of `Char` values, and hence is synonym to `{#Char}`. In Haskell, `String` is synonym to a list of `Char` values.

Clean array values can be created in several ways:

```clean
// Clean
zeroes :: Int -> .(a Int) | Array a Int
zeroes n = createArray n 0
```
fibs10 :: .(a Int) | Array a Int
fibs10 = \{1, 1, 2, 3, 5, 8, 13, 21, 34, 55\}

fibsn :: Int -> .(a Int) | Array a Int
fibsn n = {fib i \ \ i <- [0..n - 1]}

All of these functions create an array of the type .(a Int) | Array a Int, where the . indicates that the array can be updated destructively. Here, zeroes n creates an array, via the Array type class member function createArray, containing n zeroes, fibs10 contains the first ten Fibonacci numbers, and fibsn n uses an array comprehension to construct the first n Fibonacci numbers using some ordinary fib function. It should be noted that usually the programmer decides for one particular array type (lazy, strict, or unboxed) for efficiency reasons, and uses overloaded versions typically for array libraries.

Arrays can be updated destructively. The notation is very similar to record updates, but instead of a field label, an index is provided. So, with a an array, then \{a & [i] = x, [j] = y\} destructively updates a at index positions (starting at zero) i and j with values x and y respectively. Array updates can be combined concisely with array comprehensions. For instance, the function fibsn is defined more efficiently using a lazy array:

// Clean
fibsn :: Int -> {Int}
fibsn n = a
  where
    a = {createArray n 1 & [i] = a.[i - 1] + a.[i - 2]
        \ \ i <- [2..n - 1]}

Here, a.[i] selects the element at index i in array a.

Indexes can also be used in patterns, making these either constants or variables. As an example, here is a palindrome checker for arrays:

// Clean
isPalindrome :: {e} -> Bool | Eq e
isPalindrome a = size a <= 1 || check (0, size a - 1) a
  where
    check (i, j) a=:\{[i] = x, [j] = y\} =
      i >= j || x == y && check (i + 1, j - 1) a

Haskell provides only immutable arrays via the standard module Array. Arrays are implemented as an abstract datatype Array a b where a is the type of the bounds of the array and must be an instance of the Ix type class, and b is the element type. Haskell lacks denotations for arrays, array patterns, and array selections. Arrays are created using two library functions:

-- Haskell
array    :: Ix a => (a, a) -> [(a, b)] -> Array a b
listArray :: Ix a => (a, a) -> [b] -> Array a b
In both cases, the first argument \((l, u)\) defines the bounds of the array and the second argument influences the initial array elements. For `array`, each \((i, x)\) in the (finite) list updates the array at index position \(i\) to value \(x\). For `listArray`, the first \(u - 1 + 1\) entries from the (possibly infinite) list determine the initial values of the array. In both cases unaddressed positions are initialised with \(⊥\).

The `//` operator creates a new array from an existing array:

```haskell
-- Haskell
(/\) :: Ix a => Array a b -> [(a, b)] -> Array a b
```

The result array is identical to the source array, except that each \((i, x)\) in the list sets the value at index position \(i\) to \(x\).

**Exchange**

The `Array` module has been implemented in Haskell\(^*\) and can be used in Clean\(^*\). Haskell\(^*\) can import Clean arrays and manipulate them with the functions from the `StdArray` module. The Clean syntax of array element selection \(a.[i]\) conflicts with Haskell function composition and list notation. Hence, this is not supported in Haskell\(^*\). Instead, elements are selected with `a?[i]` which selects the element at index position \(i\) and returns the unaltered array `a`. Alternatively, the `Array` type class member function `select` can be used. Also, we can denote Clean arrays in Haskell\(^*\). For instance, \(\{1, 2, 3\}\), \(\{!1, 2, 3\}\), and \(\#1, 2, 3\) are legal denotations in Haskell\(^*\).

**Implementation**

Haskell arrays in the `Array` module are implemented as strict Clean arrays:

```haskell
-- Haskell*
data Array a b = Array !(!a, !a) ![b]
```

Due to this strict representation of arrays, all array operations come with strict arguments. Specialised versions of type `Int` are generated and exported, using `special` as discussed in Section 2.6, for the array operations that are overloaded in the `Ix` type class. As an example, here are the exported signatures of `array` and `listArray`:

```haskell
-- Haskell*
array :: Ix a =>
        !(!a, !a) -> ![[(a, b)]] -> Array a b special a = Int
listArray :: Ix a =>
            !(!a, !a) -> ![b] -> Array a b special a = Int
```

Also, a distinction is made between arrays that have a zero lower bound and other lower bound values.
2.11 Dynamic typing

Clean supports dynamic typing to wrap values into a black box together with its type, deferring type checking until run time. Haskell has no such feature, but GHC offers the Data.Dynamic library for similar but limited purposes. In Clean, a value is wrapped in a dynamic value using the keyword \texttt{dynamic}:

\begin{verbatim}
// Clean
wrapInt :: Int -> Dynamic
wrapInt x = dynamic x :: Int
\end{verbatim}

The type annotation is only required when the type cannot be inferred. Unwrapping a value is performed via pattern matching and specifying the expected type:

\begin{verbatim}
// Clean
unwrapInt :: Dynamic -> Int
unwrapInt (x :: Int) = x
unwrapInt (xs :: [a]) = length xs
unwrapInt ((f, x) :: (a -> Int, a)) = f x
unwrapInt (f :: A.a: [a] -> Int) = f [1..10]
unwrapInt _ = 10
\end{verbatim}

In the second and third arm, \texttt{a} is a pattern variable and is unified with the concrete type that is stored in the dynamic value. Multiple occurrences of the pattern variable in the third arm forces unification of the components of the tuple type. In the fourth arm, \texttt{a} is universally quantified, and hence the value must be a polymorphic function on lists.

Any value can be (un)wrapped, as long as there is a value representation of its type available. This is guarded by the built-in type class \texttt{TC}. For example, consider the following universal wrapping function:

\begin{verbatim}
// Clean
wrap :: a -> Dynamic | TC a
wrap x = dynamic x
\end{verbatim}

The context in which this function is used determines the type that is stored in the dynamic value. Analogously, unwrapping a value can depend on the type that the context requires:

\begin{verbatim}
// Clean
unwrap :: Dynamic -> Maybe a | TC a
unwrap (x :: a^) = Just x
unwrap _ = Nothing
\end{verbatim}

Here, the type of the context determines with which type the dynamic content is unified. This is indicated by postfixing a type pattern variable with ^, which ‘connects’ it with the type variable occurring in the type of function.
Exchange

Since Haskell does not support dynamic typing like Clean, we only have to consider the effects of Clean's dynamic typing in Haskell*. The type `Dynamic` and type class `TC` are imported via the module `StdDynamic` in Haskell* since these are built-in. When a Clean function is used that returns a dynamic value, the Haskell* module has to be able to denote such values. Therefore, it supports the keyword `dynamic`. For instance, we are able to define the `wrap` function in Haskell* as follows:

```
-- Haskell*
wrap :: TC a => a -> Dynamic
wrap x = `dynamic` x
```

The keyword is escaped using a ‘ to avoid any naming conflicts with similarly named definitions in Haskell. Also, we can unwrap a value in a dynamic pattern match in Haskell*:

```
-- Haskell*
unwrap :: TC a => Dynamic -> Maybe a
unwrap (x :: a^) = Just x
unwrap _             = Nothing
```

Implementation

Since the Clean compiler already supports dynamic typing, the implementation did not pose many challenges. The only issue raised in the Haskell parser was due to the use of the :: annotation which is obligatory when wrapping polymorphic values. It conflicts with Haskell where any expression can be annotated with a type using the same notation. For example, consider the following expression:

```
-- Haskell*
wrappedId :: Dynamic
wrappedId = `dynamic` (\x -> x) :: A.a: a -> a
```

Here, it is unclear whether the type annotation is part of Clean’s dynamic type system or Haskell’s expression. Whenever the parser recognises the keyword `dynamic`, the subsequent type annotation is part of the dynamic value. Otherwise, the type annotation is part of the expression.

2.12 Generic functions

Clean supports generic programming as advocated by Hinze (2000a) which was adopted in Clean in 2001. The style of programming is very similar to Generic Haskell (Löh et al., 2003). Generic programming is used to avoid boilerplate programming, for essentially the same purpose as instances can be derived automatically for type classes in Haskell, as discussed in Section 2.6. Haskell has no language support for generic functions.

A generic function is a recipe that is defined in terms of the structure of datatypes, rather than the datatypes themselves. The key advantage is that there are
only a few structural elements from which all custom datatypes can be constructed. For algebraic datatypes, the programmer needs to distinguish alternatives, products of (empty) fields, and basic types. As an example, here is an excerpt of the generic definition of equality:

```
// Clean
generic geq a :: a a -> Bool
geq{|Int|} x y = x == y
geq{|UNIT|} UNIT UNIT = True
geq{|EITHER|} fx _ (LEFT x1) (LEFT x2) = fx x1 x2
geq{|EITHER|} _ fy (RIGHT y1) (RIGHT y2) = fy y1 y2
geq{|EITHER|} _ _ _ _ = False
geq{|PAIR|} fx fy (PAIR x1 y1) (PAIR x2 y2) = fx x1 x2 && fy y1 y2
```

Note that this is not a single function definition, but rather a collection of function definitions that are indexed by a type constructor. They also do not need to reside in the same module, but can be defined anywhere provided that the generic type signature is in scope.

If the programmer wishes to have an instance of equality for her custom type, say `GenTree` and `Stream` defined in Section 2.9, then this is expressed as:

```
// Clean
derive geq GenTree, Stream
```

Such derived functions are exported in the same fashion.

A kind annotation is always provided for a generic function. For instance, if we wish to test some trees `x` and `y` for equality, we write `geq{|*|} x y`. Naturally, overloaded equality can be defined as a synonym of the generic variant:

```
// Clean
instance Eq (GenTree a) | geq{|*|} a where
  x == y = geq{|*|} x y
```

The programmer can deviate from the generic recipe if she wishes. In that case, the generic function has to be specialised for that specific type. Suppose that two general trees are identical if they have the same elements when visiting the tree in left-first depth-first order:

```
// Clean
geq{|GenTree|} fx x1 y2 =
  length e1 == length e2 && (zipWith fx e1 e2)
  where
    (e1, e2) = (elts x1, elts y2)
    elts {elt, kids} = [elt : concatMap elts kids]
```

The `fx` parameter is provided by the generic mechanism and is the generic equality for the element types of the generalised tree. This specialisation is exported using the `derive` syntax.
Exchange

Haskell does not have any built-in support for generic functions, therefore, we only consider using Clean’s generic functions in Haskell*. Since every use of a generic function requires a kind annotation, Haskell* supports such annotations. When importing a generic function like \texttt{geq} in a Haskell* module, an instance for a Haskell* datatype is derived using the keyword \texttt{derive}. For similar reasons as \texttt{dynamic} in Section 2.11, this keyword is escaped:

\begin{verbatim}
-- Haskell*
data BinTree a = Leaf a | Node (BinTree a) a (BinTree a)

'derive geq BinTree
\end{verbatim}

We are even able to define generic functions in Haskell*. The earlier definition of \texttt{geq} remains the same, only its signature changes:

\begin{verbatim}
-- Haskell*
'generic geq a :: a -> a -> Bool
\end{verbatim}

An escaped keyword is now used and the type no longer reflects the arity of its definition. Exporting generic functions and their derivations from a Haskell* module is not yet implemented.

Implementation

The implementation did not pose any challenges since Clean already includes support for generic functions.

2.13 Performance

Although the implementation of the front end is not yet complete, it is already possible to compile a large class of Haskell programs into efficient code. We have compared the current implementation of the double-edged front end for the Clean compiler with GHC 6.12.2 by running the complete Haskell benchmark programs of Hartel (1993). We modified the \texttt{parstof} program slightly to prevent GHC from optimising the program. It is intended that the computation is performed 40 times instead of once. To obtain good measurable execution times some of the input sizes of the programs were increased. Our benchmark environment used IA32 code on a computer with an AMD Opteron 146 2Ghz processor running the Windows XP X64 operating system. The results are shown in Table 2.1.

The columns show the name of the program, the execution times in seconds (elapsed wall clock time including startup), the ratio of execution times (comparing the execution time of GHC executables to the front end executables), and the provided options for the generated executables. For the front end we specify what garbage collector was used to obtain the best performance (‘c’ is the combination of a copying and compacting collector and ‘m’ is the combination of a marking and compacting collector) and the maximum heap size. With GHC we used the ‘-O’ optimisation flag and for the executables that required larger heaps we used
2.14 Related work

Already in Fortran, the first programming language that offered functions, it was realised that it is sometimes convenient to use foreign functions, for instance to improve efficiency by directly using assembly functions. Soon after other languages were introduced, there was the desire to use parts of other programs. There are many programming languages that offer such interpretability, usually realised by a foreign function interface. A typical interface offers a possibility to annotate a function as `external`. Then, the compiler assumes that the external function exists. It is the task of the linker to include that external function, which is compiled by the compiler of its host language, to the code generated for the program.

It is evident that this approach to exchange sources between languages imposes huge restrictions on the compiler as well as the language. Not only must the stack layout of both languages be identical, but also the memory layout of all datastructures used. For instance, both languages must use the same precision for integers, and layout for records and multidimensional arrays. An example of

<table>
<thead>
<tr>
<th>Program</th>
<th>Execution (s)</th>
<th>Front end</th>
<th>GHC</th>
<th>Ratio</th>
<th>Front end</th>
<th>GHC -O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Front end</td>
<td></td>
<td></td>
<td></td>
<td>GC</td>
<td>Heap</td>
</tr>
<tr>
<td>complab</td>
<td>0.81</td>
<td>1.03</td>
<td>0.79</td>
<td></td>
<td>c</td>
<td>8M</td>
</tr>
<tr>
<td>event</td>
<td>0.64</td>
<td>1.23</td>
<td>0.52</td>
<td></td>
<td>c</td>
<td>32M</td>
</tr>
<tr>
<td>fft</td>
<td>0.36</td>
<td>0.78</td>
<td>0.46</td>
<td></td>
<td>c</td>
<td>64M</td>
</tr>
<tr>
<td>genfft</td>
<td>0.72</td>
<td>1.37</td>
<td>0.53</td>
<td>m</td>
<td>400K</td>
<td></td>
</tr>
<tr>
<td>ida</td>
<td>0.84</td>
<td>0.87</td>
<td>0.97</td>
<td>c</td>
<td>16M</td>
<td>16M</td>
</tr>
<tr>
<td>listcompr</td>
<td>0.11</td>
<td>0.25</td>
<td>0.44</td>
<td>m</td>
<td>400K</td>
<td></td>
</tr>
<tr>
<td>listcopy</td>
<td>0.11</td>
<td>0.26</td>
<td>0.42</td>
<td>m</td>
<td>400K</td>
<td></td>
</tr>
<tr>
<td>parstof</td>
<td>0.23</td>
<td>0.19</td>
<td>1.21</td>
<td>m</td>
<td>8M</td>
<td>8M</td>
</tr>
<tr>
<td>sched</td>
<td>2.78</td>
<td>1.84</td>
<td>1.51</td>
<td>m</td>
<td>12M</td>
<td></td>
</tr>
<tr>
<td>solid</td>
<td>0.81</td>
<td>1.11</td>
<td>0.73</td>
<td>c</td>
<td>4M</td>
<td>4M</td>
</tr>
<tr>
<td>transform</td>
<td>0.91</td>
<td>1.28</td>
<td>0.71</td>
<td>m</td>
<td>400K</td>
<td></td>
</tr>
<tr>
<td>typecheck</td>
<td>0.77</td>
<td>0.86</td>
<td>0.90</td>
<td>m</td>
<td>400K</td>
<td></td>
</tr>
<tr>
<td>wang</td>
<td>0.55</td>
<td>0.64</td>
<td>0.86</td>
<td>m</td>
<td>100M</td>
<td>100M</td>
</tr>
<tr>
<td>wave4</td>
<td>0.53</td>
<td>0.72</td>
<td>0.74</td>
<td>m</td>
<td>10M</td>
<td>10M</td>
</tr>
</tbody>
</table>

Table 2.1: Execution times of Haskell using the front end and GHC

the ‘-H’ flag with the same heap size as for the Clean executables for the GHC executables, but only if this improved the performance.

All benchmarks are single-module Haskell programs. Hence, GHC cannot obtain an advantage by cross-module optimisation over our compiler. Since the current implementation of the front end is work in progress, not all planned optimisations are implemented yet. When these optimisations are implemented we will study the benchmarks and the reasons behind the observed differences. Currently, the benchmarks just show that our compiler achieves competitive results.
an issue in the interface is that Fortran starts array indices by one, while most modern languages start array indices at zero. Moreover, the array dimensions in Fortran are reversed compared to languages like C. Hence, the Fortran array declaration \( A(n, m) \) matches \( A[m][n] \) in C. The Fortran element \( A(i, j) \) matches \( A[j - 1][i - 1] \) in C. To overcome such kind of problems, many languages offer interface types which mimic their counterpart in the external language.

Both Haskell (Chakravarty, 2003) and Clean offer the possibility to exchange sources with C. Moreover, both languages offer support for using functions via this interface, GreenCard for Haskell and HtoClean for Clean. Exchanging sources between Clean and Haskell via this interface is very unattractive. The interface puts severe restrictions on the types that can be used. For instance, there is no notion of type classes and higher-order functions, and parameterised recursive datatypes cause all kinds of problems. Also, such an interface is completely unsuited for lazy evaluation since this is not supported by C.

Since C is a subset of the C++, every valid C program is also valid C++. Hence, every compiler for C++ accepts C, which makes interoperability between these two languages very easy. Such an approach is not applicable for our purposes since Clean nor Haskell is a subset of the other.

The Microsoft .NET Framework supports multiple programming languages and focuses on language interoperability. It contains special designed languages like C#, F# and J#, as well as support for standard languages like Python and Lisp. Some alternative and free implementations of parts of this framework are Mono, CrossNet and Portable.NET. Since Haskell nor Clean is designed for such a framework this approach is not suited for our needs. Moreover, these frameworks are based on an object-oriented view of the world and have limited support for features in modern lazy functional languages.

There is some work to translate Haskell to Clean in order to obtain Haskell programs with the speed of Clean programs. First, Hegedus (2001) translated Haskell structures to Clean. Next, Diviánszky (2003) implemented a partial compiler from Haskell to Clean based on these concepts. Hackle (Naylor, 2004) is a compiler from a restricted subset of Haskell 98 to Clean. This compiler actually achieved performance gain compared to GHC for a number of programs. Although each of these approaches studied translating Haskell to Clean, the exchange of language features from both languages was not considered.

There are a number of stand-alone Haskell implementations. The flagship compiler GHC supports the complete Haskell 98 standard, as well as a wide variety of language extensions. Hugs 98 provides an almost complete implementation of the standard, but unfortunately the last release dates from 2006. Nhc 98 is a small compiler that is compliant to the standard, its last release stems from 2007. Yhc branched from Nhc 98, but is not yet a complete Haskell 98 compiler. The recent UHC supports almost the complete standard and adds several experimental language extensions. None of these has support for interoperability with Clean.

### 2.15 Conclusion

In this chapter we have described what it takes to exchange sources between Clean and Haskell. We discussed most of the differences in language features and the
required extensions of both Clean and Haskell to denote them. This has resulted in
two dialects, dubbed Clean* and Haskell* respectively. Also, we briefly explained
how their exchange is facilitated in a concrete implementation. We have seen how
some of the language features go together nicely hand-in-hand (e.g., uniqueness
typing and monads), while others lead to subtle conflicts (e.g., records).

Besides the exchange of sources, the front end supports the exchange of features
to a certain extent as well. Clean programmers can use constructs like newtypes.
Haskell programmers can now use uniqueness typing, dynamic typing, and generic
functions. Additionally, the front end comes with benefits for both Haskell and
Clean programmers. For instance, Haskell programmers can use the full-fledged
IDE including project manager. Also, performance of compiled Haskell programs
looks promising: on a par and for computation-intensive applications often slightly
better than GHC. For Clean programmers, it is nice that their work becomes more
easily accessible to the large Haskell community.

Although the most important features of Haskell 98 have been implemented,
the list of remaining issues is still rather long since some features took much more
work than expected. When we started this project about three years ago, we knew
that Haskell is a more baroque language than Clean. But only after digging into
the details of the language we discovered that Haskell was even more complicated
than anticipated. For instance, since Haskell makes heavily use of overloading and
monads, more effort was needed to retain the efficiency that Clean is well known
for. Also, the number of Haskell libraries which are really Haskell 98 compliant
is rather limited. To enable the practical reuse of Haskell libraries, we have to
implement some of GHC’s extensions, such as generalised algebraic datatypes and
type families. This is challenging, not only in terms of the programming effort,
but more because of the consequences it will have on features such as uniqueness
typing. We believe this double-edged front end provides an excellent research and
implementation laboratory to investigate these avenues.
Part II

The necessity of dynamic typing
A lightweight approach to datatype-generic rewriting

Abstract

Term-rewriting systems can be expressed as generic programs parameterised over the shape of the terms being rewritten. Previous implementations of generic rewriting libraries require users to either adapt the datatypes that are used to describe these terms or to specify rewrite rules as functions. These are fundamental limitations: the former implies a lot of work for the user, while the latter makes it hard, if not impossible, to document, test, and analyse rewrite rules. In this chapter, we demonstrate how to overcome these limitations by making essential use of type-indexed datatypes. Our approach is lightweight in that it is entirely expressible in Haskell using generalised algebraic datatypes and type families and can be readily packaged for use with contemporary Haskell distributions.

3.1 Introduction

Consider a Haskell datatype \texttt{Prop} for representing formulae of propositional logic:

\begin{verbatim}
data Prop = Var String | T | F | Not Prop | Prop :\&: Prop | Prop :\lor: Prop
\end{verbatim}

Suppose we wish to simplify such formulae using the principle of contradiction \( p \land \neg p \rightarrow \bot \). Ideally, our formulation of this rewrite rule as an executable program is neither much longer nor much more complicated than this rule itself.

One approach is to encode the rule as a function and then to apply it to individual formulae using some bottom-up traversal combinator \texttt{transform}:

\begin{verbatim}
simplify :: Prop -> Prop
simplify = transform contradiction
where
  contradiction (p :\&: Not q) | p \equiv q = F
  contradiction p = p
\end{verbatim}

Although this implementation is relatively straightforward, encoding rules by functions has a number of drawbacks. To start with, rules cannot be concise one-line definitions as we have to provide a catch-all case in order to avoid pattern-matching failures at run-time. Secondly, pattern guards (such as \( p \equiv q \) in our example) are
needed to deal with multiple occurrences of variables, cluttering the definition. Lastly, rules cannot be analysed easily since it is impossible to inspect functions.

A way to overcome these drawbacks is to provide specialised rewriting functionality. That is, we can define a datatype representing rewrite rules on formulae and implement the machinery required for rewriting (e.g., functions for matching formulae against rules and substituting formulae for metavariables) on top of this datatype. While this does overcome the drawbacks mentioned above, this approach comes with a serious disadvantage: it requires a large amount of datatype-specific code to be written. If our next task is to rewrite, say, arithmetic expressions, we have to define a new datatype for representing rewrite rules and a new implementation of all the rewriting machinery.

However, both the datatype for representing rules and the associated rewriting machinery can be determined from the type that is used to describe the terms being rewritten. Hence, there is an excellent opportunity for datatype-generic programming here. In this chapter, we seize this opportunity and present a rewriting library that is generic in the type of terms being rewritten. Using our library, the example above can be written as:

\[
\text{simplify} :: Prop \rightarrow Prop \\
\text{simplify} = \text{transform} \ (\text{rewriteWith contradiction}) \\
\text{where} \\
\text{contradiction} = \text{synthesise} \ (\lambda p \rightarrow p \land \lnot p \mapsto \bot)
\]

The library provides \text{rewriteWith}, \text{synthesise}, and \mapsto, which are generic and, in this case, instantiated with the type of propositional formulae \text{Prop}. A noticeable aspect of our approach is that metavariables in rewrite rules, such as \( p \) in our example, are introduced through ordinary function abstraction in Haskell, allowing the user to define her rules in terms of the term type \text{Prop} rather than some dedicated type for representing rules over \text{Prop}. The body of the function \text{contradiction} is now a fairly direct transcript of the rule \( p \land \lnot p \rightarrow \bot \). As we will see, rewrite rules constructed with our library neither suffer from the drawbacks of the approach that uses pattern matching nor require large amounts of datatype-specific boilerplate code.

More specifically, the contributions of this chapter are the following:

- We present a library\(^1\) for term rewriting that is implemented using a simple design pattern (Section 3.4) for datatype-generic programming in Haskell extended with type families (Chakravarty et al., 2005a & b; Schrijvers et al., 2008). As such, our library is ‘lightweight’ and can be used readily with recent versions of the flagship Haskell compiler GHC.

- To represent rewrite rules, our library needs to extend the type that is used to describe the terms being rewritten with an extra constructor for metavariables internally (Section 3.5.2). This extension is constructed generically using a type-indexed datatype (Hinze et al., 2004). Distinct metavariables in a single rewrite rule can, in our approach, range over rewritable terms of different type (Section 3.5.1).

\(^1\)The library is dubbed \text{guarded-rewriting} and available on Hackage.
3.2 Representing rewrite rules

- Internally, the library implements rewriting in terms of generic functions for pattern matching (Section 3.5.4) and substitution (Section 3.5.3) over generically extended datatypes. These datatypes are, however, completely hidden from the user, who formulates her rewrite rules using the constructors of the types of terms that are to be rewritten (Section 3.6).

- We compare the performance of our library to that of other approaches to term rewriting in Haskell (Section 3.10).

The remainder of this chapter is structured as follows. In Section 3.2, we discuss two fundamental approaches to representing rewrite rules in Haskell. In Section 3.3 we present our proposal for a datatype-generic library for term rewriting from a user’s perspective.

Sections 3.4 to 3.6 deal with the implementation of our library’s main functionality. Section 3.4 showcases, through an example generic function, how datatype-generic functions are implemented in our library. Section 3.5 discusses how generic rewriting functionality is composed from more elementary generic functions for pattern matching and substitution, and shows how these functions are implemented. In Section 3.6, we demonstrate how the not so programmer-friendly representation of rewrite rules, used internally by the generic functions from Section 3.5, is hidden from the users of our library.

Sections 3.7 and 3.8 discuss additions to the core functionality. In Section 3.7, it is shown how nonsensical rewrite rules can be detected statically (i.e., without applying them). In Section 3.8, the library is extended with support for rewrite rules that have preconditions associated with them.

Section 3.9 discusses, as a case study, the use of our library in a realistic application. Section 3.10 presents the results of two performance benchmarks. Section 3.11 discusses related work and Section 3.12 concludes.

3.2 Representing rewrite rules

Before we present our approach to datatype-generic rewriting in Section 3.3, let us first have a more in-depth look at the two fundamental approaches to representing rewrite rules in Haskell: the intensional approach (Section 3.2.1) and the extensional approach (Section 3.2.2).

3.2.1 Intensional representations

The intensional approach to representing rewrite rules encodes rules as Haskell functions, using pattern matching to check whether the argument term matches the left-hand side of the rule. If this is indeed the case, the right-hand side of the rule is returned; otherwise, the argument term is returned unchanged. For example, the rule \( \neg(p \land q) \rightarrow \neg p \lor \neg q \), one of the De Morgan’s laws, is intensionally encoded as follows:

\[
\begin{align*}
\text{deMorgan} :: & \text{Prop} \rightarrow \text{Prop} \\
\text{deMorgan} (\text{Not} (p \land q)) &= \text{Not} p \lor \text{Not} q \\
\text{deMorgan} p &= p
\end{align*}
\]
Note that the last line prevents arguments that do not match the pattern $\neg(p \land q)$ from causing run-time errors.

As Haskell lacks support for nonlinear patterns, rewrite rules containing metavariables with multiple left-hand-side occurrences cannot be written as functions directly. Instead, such variables are encoded by means of so-called pattern guards. For instance, a rule for the principle of the excluded middle, $p \lor \neg p \rightarrow \top$, in which the metavariable $p$ occurs twice at the left-hand side, is implemented by the following function:

\[
\text{excludedMiddle} :: \text{Prop} \rightarrow \text{Prop} \\
\text{excludedMiddle} (p :\lor: \text{Not } q) \mid p \equiv q = T = p
\]

where the second occurrence of $p$ is replaced by an occurrence of a fresh variable $q$ and equality of $p$ and $q$ is enforced through the guard $p \equiv q$. Note that this encoding requires equality to be defined for values of type Prop.

In some applications of rewriting, it is useful to know whether or not a rewrite rule was applied successfully. This information can be made available, at the expense of some additional notational overhead, by wrapping the rewriting result in a Maybe value:

\[
\text{excludedMiddleM} :: \text{Prop} \rightarrow \text{Maybe Prop} \\
\text{excludedMiddleM} (p :\lor: \text{Not } q) \mid p \equiv q = \text{Just } T
\]

Encoding rewrite rules in terms of Haskell functions allows for function-parameterised traversal combinators to be used directly in rewriting applications. As an example, the Uniplate library (Mitchell and Runciman, 2007) provides, amongst others, the combinator transform:

\[
\text{transform} :: \text{Uniplate } \alpha \Rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha
\]

which applies its argument function in a bottom-up fashion to all recursive positions in a tree. Given a suitable Uniplate-instance for the type Prop, it is straightforward to use this combinator to remove certain classes of tautological clauses from propositional formulae:

\[
\text{removeTautologies} :: \text{Prop} \rightarrow \text{Prop} \\
\text{removeTautologies} = \text{transform excludedMiddle}
\]

However, even though Haskell’s pattern-matching facilities enable a more or less direct encoding of rewrite rules as functions and the interaction with traversal libraries comes almost for free, the intensional approach to representing rewrite rules raises some issues:

- Intensionally represented rules cannot be observed, as in Haskell it is not possible to inspect functions. Still, there are several reasons why it is desirable to have observable rewrite rules:

**Documentation** If rules are observable, they can be pretty-printed in order to generate documentation for a rewrite system.
3.2 Representing rewrite rules

**Static checking** Observability of rules allows for checking whether a given set of rewrite rules constitutes a confluent and terminating rewrite system.

**Automated testing** In most applications, a rule is expected to preserve the semantics of the term being rewritten. One way to test this property is to randomly generate terms, to rewrite these, and then to check whether the rewritten terms indeed have the same semantics as the original terms. However, a rewrite rule with a nontrivial left-hand side will most likely not match successfully against a randomly generated term. Hence, such rules are in danger of not getting tested sufficiently. If left-hand sides of rules are inspectable, term generation can be directed to produce matching terms more often, effectively improving test coverage.

**Associativity- and commutativity-aware rewriting** Various domains, such as that of logical propositions, have associative and commutative operators. If the rewriting infrastructure is aware of this fact, rewrite rules can be specified more concisely and repetition can be avoided. With an extensional approach, this can be implemented by making the matching algorithm return all possible substitutions. In an intensional approach, the behavior of pattern matching is fixed and cannot be made aware of these operators.

**Inversion** If the left-hand side and right-hand side of a rewrite rule can be accessed, these can be exchanged, resulting in the inverse of the rule.

**Tracing** When a sequence of rewrite steps leads to an unexpected result, one may want to learn which rules were applied in which order.

- It is tedious to have to specify a catch-all case when rules are encoded as functions. All rule definitions require this extra case.
- The lack of nonlinear pattern matching in Haskell becomes a nuisance if left-hand sides of rules contain many occurrences of the same variables.
- As Haskell lacks first-class patterns, the user cannot easily abstract over commonly occurring structures in the left-hand sides of rewrite rules.

These issues can be overcome by switching to an extensional representation instead.

### 3.2.2 Extensional representations

In the extensional approach, rewrite rules are not encoded as functions, but as values of a datatype, so that the left- and right-hand sides of rules become observable:

```haskell
data Rule α = Rule {lhs :: α, rhs :: α}
```

Values of type `Rule α` are used to encode rewrite rules with left- and right-hand sides of type `α`. For example, rewrite rules for formulae of propositional logic can be expressed as values of type `Rule EProp`, where `EProp` is an extended version...
of the datatype `Prop` of propositional formulae with an extra constructor `Metavar` to represent metavariable occurrences in rewrite rules:

```haskell
data EProp = EVar String | ET | EF
| ENot EProp
| Metavar String
```

With values of type `Rule EProp` in place, we need to define rewrite functions that interpret these values as functions over propositions represented by `Prop`:

```haskell
rewritePropWith :: Rule EProp → Prop → Prop
```

Here we do not give an implementation of `rewritePropWith`, but note that its type (and thus its implementation) is specific to propositional formulae. If we want to implement rewrite functionality that works on different datatypes, then we have to define new rewrite functions for these types.

With the proposition-specific rewrite function `rewritePropWith`, rules over propositional formulae can be written and used as in the following function:

```haskell
removeTautologies :: Prop → Prop
removeTautologies = transform (rewritePropWith excludedMiddle)
where
excludedMiddle = Rule { lhs = Metavar "p" #: ENot (Metavar "p") , rhs = ET }
```

An apparent inconvenience of this style of defining rules is that we cannot reuse the type `Prop` of terms being rewritten and its constructors `Not` and `∨`. Instead, to provision for metavariables, we have to use the extended representation `EProp` and its constructors `ENot` and ` #:`.

### 3.3 Datatype-generic rewriting

In this section, we present the interface to our library for datatype-generic rewriting. In Sections 3.4 to 3.6, we zoom in at the concrete implementation of this interface.

The rewrite system that we present in this chapter uses extensionally represented rewrite rules. As observed in the previous section, straightforward implementations of such rewrite systems suffer from two drawbacks: (1) they require a significant amount of datatype-specific code and (2) rewrite rules need to be expressed in terms of a new datatype obtained by extending the original datatype with a constructor for metavariables. Our system, however, is carefully designed to circumvent these drawbacks: (1) we provide a single implementation of rewriting that is generic in the type of terms being rewritten and (2) we completely hide the internal representation of rewrite rules from the user of our library.

More specifically, in our approach rewrite rules are specified in terms of templates:
3.3 Datatype-generic rewriting

`closedWorldTemplate :: Template Prop`
`closedWorldTemplate = Not T ↦→ F`

`contradictionTemplate :: Prop → Template Prop`
`contradictionTemplate p = p :∧: Not p ↦→ F`

`deMorganTemplate :: Prop → Prop → Template Prop`
`deMorganTemplate p q = Not (p :∧: q) ↦→ Not p :∨: Not q`

Templates are constructed by means of an operator ↦→ which takes a left-hand side and a right-hand side of a type α and produces a template for rewrite rules on α:

`(~→) :: α → α → Template α`

Note that both sides of a template are just values of the type of terms being rewritten. In particular, templates are expressed without need for an additional datatype providing for metavariables. Instead, metavariables are encoded as ordinary Haskell function arguments. The template for the De Morgan rule from the example above, for instance, uses two metavariables which are introduced through function arguments `p` and `q`.

To prepare templates for use in our rewrite system, the user needs to synthesise rules from these. To this end, the library provides an overloaded function `synthesise` (defined in Section 3.6.3), that takes templates or functions producing templates for rewrite rules on some type α to values of type `Rule α`:

`closedWorld, contradiction, deMorgan :: Rule Prop`
`closedWorld = synthesise closedWorldTemplate`
`contradiction = synthesise contradictionTemplate`
`deMorgan = synthesise deMorganTemplate`

Here, values of type `Rule α` (with an implementation that differs slightly from the one given above, as will be discussed in Section 3.5) form the internal representation of rewrite rules on α in our library.

The generic rewrite functionality is now exposed through a pair of rewrite functions `rewriteWith` and `rewriteWithM`. The first function takes as arguments a rule over some rewritable type α (as we will see in Section 3.3.2) and a value of type α, and attempts to apply the rule to the value:

`rewriteWith :: Rewritable α ⇒ Rule α → α → α`

For example, the following expression yields `F`:

`rewriteWith closedWorld (Not T)`

If the second argument to `rewriteWith` does not match the left-hand side of its first argument, the value to be rewritten is returned unmodified. For instance, consider the following expression:

`rewriteWith contradiction (Var "x" :∧: Not (Var "y"))`

This yields `Var "x" :∧: Not (Var "y")` as the argument term does not match a contradictory formula. To make a failed attempt at rewriting explicit in the value returned, a second generic rewrite function wraps its result in a monad µ:
A lightweight approach to datatype-generic rewriting

\[ \text{rewriteWithM} :: (\text{Rewritable } \alpha, \text{Monad } \mu) \Rightarrow \text{Rule } \alpha \rightarrow \alpha \rightarrow \mu \alpha \]

For example, instantiating \( \mu \) with the \textit{Maybe}-monad, the following expression results in \textit{Nothing}:

\[ \text{rewriteWithM \ deMorgan \ (} T :\land : F) \]

However, the next expression yields \textit{Just} \((\text{Not } T :\lor : \text{Not } F)\):

\[ \text{rewriteWithM \ deMorgan \ (} \text{Not} \ (T :\land : F)) \]

As with other lightweight approaches to generic rewriting, such as Scrap Your Boilerplate (L"ammel and Peyton Jones, 2003) and Uniplate (Mitchell and Runciman, 2007), a small effort is required from the users of our library in order to prepare their datatypes for generic rewriting. In particular, they must make their datatypes instances of the type classes \textit{Representable} (Section 3.3.1) and \textit{Rewritable} (Section 3.3.2).

### 3.3.1 Representing the structure of datatypes

In our library, the structure of datatypes is described through instances of a type class \textit{Representable}:

```haskell
class Representable \alpha where
  type Rep \alpha :: *
  from :: \alpha \rightarrow Rep \alpha
  to :: Rep \alpha \rightarrow \alpha
```

Here, \( \text{Rep} \) is a so-called associated type synonym (Chakravarty et al., 2005a). A type \( \alpha \) is representable if it is isomorphic to its generic representation type \( \text{Rep} \alpha \); the isomorphism is witnessed by a pair of functions \( \text{from} \) and \( \text{to} \) that convert between the type and its generic representation.

Base types, such as \textit{Int}, \textit{Float}, and \textit{Char} form their own generic representations:

```haskell
instance Representable Int where
  type Rep Int = Int; from = id; to = id

instance Representable Float where
  type Rep Float = Float; from = id; to = id

instance Representable Char where
  type Rep Char = Char; from = id; to = id
```

Further generic representation types are composed from a fixed set of structure constructors. These include the nullary type constructor \textit{Nil} and the binary type constructors \textit{:+} and \textit{:::}, defined as follows:

```haskell
infixr 6 :+:;
infixr 5 :::;
```
3.3 Datatype-generic rewriting

\[
\begin{align*}
\textbf{data} \ Nil & = \ Nil \\
\textbf{data} \ \alpha \ :*: \ \beta = \ \text{Inl} \ \alpha \ | \ \text{Inr} \ \beta \\
\textbf{data} \ \alpha \ :::: \ \beta = \ \alpha \ :::: \ \beta
\end{align*}
\]

A given datatype’s representation type follows immediately from its structure. Choice amongst data constructors is encoded in terms of right-nested sums constructed by \( :* \). A data constructor itself is represented as a type-level list of its argument types, constructed by \( :::: \) and \( \text{Nil} \). Note that, instead of the more common sums-of-products representation of datatypes (Jansson and Jeuring, 1997; Backhouse et al., 1999; Hinze, 2000b), we use a list-like representation (Hodermans et al., 2006) as we want to make sure that constructor arguments are always encoded as the first operand of the constructor \( :::: \). For example, consider Haskell’s \( \text{Maybe} \)-type:

\[
\textbf{data} \ \text{Maybe} \ \alpha = \ \text{Nothing} \ | \ \text{Just} \ \alpha
\]

This datatype is represented by the type \( \text{Nil} :* (\alpha :::: \text{Nil}) \) and we can write the following instance:

\[
\textbf{instance} \ \text{Representable} \ (\text{Maybe} \ \alpha) \ \text{where} \\
\textbf{type} \Rep (\text{Maybe} \ \alpha) = \text{Nil} :* (\alpha :::: \text{Nil})
\]

from \( \text{Nothing} \) = \text{Inl} \ \text{Nil}
from \( \text{Just} \ x \) = \text{Inr} \ (x :::: \text{Nil})
to \( \text{Inl} \ \text{Nil} \) = \text{Nothing}
to \( \text{Inr} \ (x :::: \text{Nil}) \) = \text{Just} \ x

The member functions \( \text{from} \) and \( \text{to} \) form a so-called embedding-projection pair and (are supposed to) witness the isomorphism between a type and its generic representation ‘modulo undefinedness’. Thus, it should hold that \( \text{to} \circ \text{from} = \text{id} \) and \( \text{from} \circ \text{to} \subseteq \text{id} \) (Hinze, 2000b).

Another example is the functional programmer’s all-time favourite datatype, namely, lists:

\[
\textbf{data} \ [\alpha] = [\ ] \mid \alpha : [\alpha]
\]

This datatype is represented in our approach by \( \text{Nil} :* (\alpha :::: [\alpha] :::: \text{Nil}) \), yielding the following declaration:

\[
\textbf{instance} \ \text{Representable} \ [\alpha] \ \text{where} \\
\textbf{type} \Rep [\alpha] = \text{Nil} :* (\alpha :::: [\alpha] :::: \text{Nil})
\]

from \( [\ ] \) = \text{Inl} \ \text{Nil}
from \( \ (x : xs) \) = \text{Inr} \ (x :::: xs :::: \text{Nil})
to \( \text{Inl} \ \text{Nil} \) = \( [\ ] \)
to \( \text{Inr} \ (x :::: xs :::: \text{Nil}) \) = \( x : xs \)

Note that the generic representation types of recursive datatypes are themselves nonrecursive: \( \text{from} \) only converts the top-level constructor of a value into its generic representation and leaves all subtrees untouched.

49
Recall the type \textit{Prop} of propositional formulae from Section 3.1:

\begin{verbatim}
data Prop = Var String | T | F | Not Prop | Prop ∧ Prop | Prop ∨ Prop
\end{verbatim}

First, we define abbreviations for the generic representations of the alternatives:

\begin{verbatim}
type Var = String ::: Nil
type T = Nil

ty
pe Not = Prop ::: Nil
type And = Prop ::: Prop ::: Nil
type Or = Prop ::: Prop ::: Nil
\end{verbatim}

Then, the corresponding instance is defined as follows:

\begin{verbatim}
instance Representable Prop where
from (Var x) = Inl (x ::: Nil)
from T = Inr (Inl Nil)
from F = Inr (Inr (Inl Nil))
from (Not p) = Inr (Inr (Inl (p ::: Nil)))
from (p :∧: q) = Inr (Inr (Inl (p ::: q ::: Nil)))
from (p :∨: q) = Inr (Inr (Inr (Inr (Inl (p ::: q ::: Nil)))))
to (Inl (x ::: Nil)) = Var x
to (Inr (Inl Nil)) = T

to (Inr (Inr (Inl Nil))) = Not p
to (Inr (Inr (Inr (Inl (p ::: Nil))))) = p :∧: q
to (Inr (Inr (Inr (Inr (Inl (p ::: q ::: Nil))))) = p :∨: q
\end{verbatim}

Instance declarations of \textit{Representable} can be quite verbose, as in the case for \textit{Prop}. However, these declarations are completely determined by the structure of the represented datatypes and can easily be derived automatically, for example by means of a Template Haskell program (Sheard and Peyton Jones, 2002). Moreover, all that needs to be done to use our library on a user-defined datatype, such as \textit{Prop}, is declaring it an instance of \textit{Representable} and \textit{Rewritable}. As we will see next, an instance of the latter can be given almost effortlessly.

\subsection{Making terms rewritable}

The type class \textit{Rewritable} of types with rewritable values is defined as follows:

\begin{verbatim}
class (Representable α,
    Typeable α,
    Eq (Rep α),
    Extensible (Rep α), Matchable (Rep α), Substitutable (Rep α),
    Sampleable (Rep α), Diffable (Rep α)) ⇒ Rewritable α
\end{verbatim}
As this type class does not have any methods or associated types, it is only introduced for its superclass constraints. These constraints encode the conditions that need to be fulfilled by a term type in order for its values to be rewritable.

Not only do we need an instance of `Representable`, we also require term types to be in the type class `Typeable` that was originally introduced for use with the Scrap Your Boilerplate-library (Lämmel and Peyton Jones, 2003). Currently, `Typeable` is Haskell’s de facto standard API for reifying types at the value level and as such it is included in the base libraries that ship with GHC. Recent versions of GHC even provide support for automatically deriving instances of `Typeable` for user-defined datatypes.

The remaining superclass constraints on `Rewritable` place restrictions on the generic representations of term types and make specific parts of the generic rewriting machinery available for all instances of `Rewritable`. More specifically, each of these constraints accounts for the existence of one generic function. As representation types are built from a limited set of type constructors, these constraints imply no additional burden on the user of our generic rewriting library. That is, all needed instances for the base types `Int`, `Float`, and `Char` and the representation constructors `Nil`, `:+:`, and `:::` are already provided by the library. The details behind these instances are discussed in the next sections: in Section 3.4 we give instances of the standard type class `Eq` for our generic representation types; in Section 3.5 we give the definitions and instances of the custom type classes `Extensible`, `Matchable`, and `Substitutable`, while Section 3.6 covers `Sampleable` and `Diffable`.

For now, we observe that putting the type of a rewritable term in the type class `Rewritable` reduces to a mere one-liner:

```haskell
instance Rewritable Int
instance Rewritable Float
instance Rewritable Char
instance Rewritable α ⇒ Rewritable (Maybe α)
instance Rewritable α ⇒ Rewritable [α]
instance Rewritable Prop
```

Of course, we also need appropriate instances of `Representable` and `Typeable` to be in place.

### 3.4 Generic equality

The previous section introduced the interface to our library for datatype-generic rewriting. Let us now turn to the concrete implementation of this interface.

In this section, we present an implementation of a type-indexed equality function. In the next section, this generic function is used in our implementation of generic pattern matching, but here it also serves as a neat example of the design pattern for lightweight type-indexed functions that we employ for all generic functions in our library. The general pattern for implementing generic functions is that we overload a given function \( f \) for all generic representation types and then derive a generic version \( f' \) that ‘ties the knot’ and works for all types in `Rewritable`. 

51
In our implementation, we rely on the type class `Eq` from Haskell’s Standard Prelude to provide an interface for overloaded equality:

```haskell
class Eq α where
    (≡), (≠) :: α → α → Bool
    x ≡ y = ¬ (x ≠ y)
    x ≠ y = ¬ (x ≡ y)
```

As the class `Rewritable` requires the generic representation types of all its instances to be in the class `Eq`, we can directly define an equality operator `≡'` that works for all types of rewritable terms:

```haskell
(≡') :: Rewritable α ⇒ α → α → Bool
x ≡' y = from x ≡ from y
```

To test two equally typed rewritable terms for equality, we convert them to their generic representations and then test these for equality.

It remains to declare instances of `Eq` for the types that appear in generic representations. The case for `Nil` is straightforward:

```haskell
instance Eq Nil where
    Nil ≡ Nil = True
```

For sums, we require the summands to be instances of `Eq` and test whether both generic representations have their origins in the same alternative. If so, both values are compared recursively; otherwise, we produce `False`:

```haskell
instance (Eq α, Eq β) ⇒ Eq (α :+: β) where
    Inl x ≡ Inl y = x ≡ y
    Inr u ≡ Inr v = u ≡ v
    _     ≡ _     = False
```

In the case for `:::` we make use of the fact that, in our encoding of a datatype’s structure, the second type argument of `:::` is always another type-level list and so we can assume that this type argument is itself in `Eq` as well. The first type argument, however, can be any type and, hence, we cannot just assume it to be an instance of `Eq`. Instead, we require this type argument to be in `Rewritable`, so that we can use the operator `≡'` defined above to compare values of this type:

```haskell
instance (Rewritable α, Eq β) ⇒ Eq (α ::: β) where
    (x ::: xs) ≡ (y ::: ys) = x ≡' y ∧ xs ≡ ys
```

### 3.5 Matching and substituting

In the previous section, we demonstrated how generic functions are implemented in our library. We continue our exploration of the internals of the library by discussing the core functionality of our library: the implementation of the function `rewriteWith` and its monadic companion `rewriteWithM`.
These are implemented in terms of two generic functions `match'` and `substitute'`:

```
match' :: (Rewritable α, Mappable Γ, Monad µ) ⇒
        Pattern Γ α → α → µ (Substitution Γ)
substitute' :: (Rewritable α, Monad µ) ⇒
           Substitution Γ → Pattern Γ α → µ α
```

The type `Pattern Γ α` from Section 3.5.2 is used in our library for the extensional representation of the left- and right-hand sides of rewrite rules over a term type `α`. Its type argument `Γ` is a so-called metavariable environment: a type-level list that encodes the types of the metavariables in a rewrite rule. Successfully matching a term against a left-hand-side pattern results in a substitution, as defined in Section 3.5.3 for the metavariables that occur in the pattern. As pattern matching may fail, the function `match'` returns its result in a monad `µ`. This function requires the metavariable environment `Γ` involved to be in the type class `Mappable`, defined in Section 3.5.1, which simply means that an empty substitution can be produced for `Γ`. Substitutions are partial maps from metavariables drawn from a given environment to matched subterms. Given such a substitution and a right-hand-side pattern, the generic function `substitute'` attempts to construct a new term value. This construction fails if the substitution is not defined for all metavariables that occur in the right-hand-side pattern, which explains the monadic result type of `substitute'`.

As metavariable environments are only of interest to the internals of our library, they are hidden from the user by wrapping the left- and right-hand-side patterns that constitute a rewrite rule in an existential type:

```
data Rule :: ⋆ → ⋆ where
          Rule :: Mappable Γ ⇒ Pattern Γ α → Pattern Γ α → Rule α
```

Here, the existential type `Rule` is defined using the syntax of so-called generalised algebraic datatypes (Peyton Jones et al., 2006).

Given the existential `Rule` and suitable definitions of `match'` and `substitute'`, the monadic rewrite function `rewriteWithM` can be written as:

```
rewriteWithM :: (Rewritable α, Monad µ) ⇒ Rule α → Rule α → µ α
rewriteWithM (Rule lhs rhs) x = do
   s ← match' lhs x
substitute' s rhs
```

That is, the term `x` is matched against the left-hand side `lhs` of a given rewrite rule. If the match is successful, the resulting substitution `s` is applied to the right-hand side `rhs` of the rewrite rule in order to produce the result term. An implementation for the nonmonadic rewrite function `rewriteWith` is obtained by instantiating the type of `rewriteWithM` with the `Maybe-monad`:

```
rewriteWith :: Rewritable α ⇒ Rule α → Rule α → α
rewriteWith rule x = case rewriteWithM rule x of
   Nothing → x
   Just y → y
```

In the remainder of this section, we discuss the implementation of typed metavariables (Section 3.5.1), generic patterns (Section 3.5.2), generic substitutions (Section 3.5.3), and generic pattern matching (Section 3.5.4).
3.5.1 Typed metavariables

In our extensional representation of rewrite rules, we encode metavariables by De Bruijn indices (De Bruijn, 1972). Our implementation allows different metavariables to range over differently typed subterms. To enforce a type-safe use of metavariables, we adopt the approach of Pasalić and Linger (2004) and implement metavariables as values of the datatype \texttt{Ref} of typed references:

\begin{verbatim}
data Ref :: ⋆ → ⋆ → ⋆ where
  RZero :: Ref (α ::: Γ) α
  RSucc :: Ref Γ α → Ref (β ::: Γ) α
\end{verbatim}

Here, we use as metavariable environments \( Γ \) the heterogeneous lists constructed from \( \text{Nil} \) and \( :::: \) that we also use in generic representations. A value of type \( \text{Ref} \ Γ \ α \) then carries the Peano encoding of an index for an \( α \)-typed position in a heterogeneous list of type \( Γ \). Note that such a value can never refer to an empty list, simply because the constructor types dictate that the lists contain at least one value.

As an example of the use of \( \text{Ref} \), consider the function \texttt{deref} for dereferencing a typed reference to a value in a heterogeneously typed list:

\begin{verbatim}
deref :: Ref Γ α → Γ → α
deref RZero (x :::: _) = x
(deref (RSucc r) (xS :::: xs)) = deref r xs
\end{verbatim}

In the implementation of \texttt{match'} and \texttt{substitute'}, typed references are used as indices into heterogeneously typed partial maps:

\begin{verbatim}
data PMap :: ⋆ → ⋆ where
  PNil :: PMap Nil
  PCons :: Rewritable α ⇒ Maybe α → PMap Γ → PMap (α ::: Γ)
\end{verbatim}

Values of type \( \text{PMap} \ Γ \) are partial maps from \( Γ \)-typed references to rewritable terms. Looking up a value in a partial map is implemented through the function \texttt{lookup}:

\begin{verbatim}
lookup :: Monad µ ⇒ Ref Γ α → PMap Γ → µ α
lookup RZero (PCons Nothing _) = fail "unbound variable"
lookup RZero (PCons (Just x) _) = return x
lookup (RSucc r) (PCons _ s) = lookup r s
\end{verbatim}

This function returns its result in a monad \( µ \) to provide for the case in which the lookup fails. Since the types of the \( RZero \) and \( RSucc \) constructors ensure that the referenced partial map is nonempty, the definition of \texttt{lookup} does not require a case for \texttt{PNil}.

For the construction of partial maps of type \( \text{PMap} \ Γ \), we require that \( Γ \) is a type-level list of rewritable-term types, so that \texttt{PNil} and \texttt{PCons} can be used to produce an initial, empty map. To this end, we make the list constructors \texttt{Nil} and \( :::: \) instances of a type class \texttt{Mappable} that provides an empty-map constructor:
3.5 Matching and substituting

```haskell
class Mappable Γ where
  empty :: PMap Γ

instance Mappable Nil where
  empty = PNil

instance (Rewritable α, Mappable Γ) ⇒ Mappable (α ::: Γ) where
  empty = PCons Nothing empty
```

Updating a rewritable term in a partial map involves recursing a typed reference and traversing the map until the appropriate position has been reached:

```haskell
update :: Ref Γ α → α → PMap Γ → PMap Γ
update RZero x (PCons _ s) = PCons (Just x) s
update (RSucc r) x (PCons mb s) = PCons mb (update r x s)
```

Singleton mappings are constructed by updating a single term in an empty map:

```haskell
singleton :: (Rewritable α, Mappable Γ) ⇒ Ref Γ α → α → PMap Γ
singleton r x = update r x empty
```

Finally, two maps for the same environment Γ can be merged if they agree on their codomain:

```haskell
(⊕) :: Monad μ ⇒ PMap Γ → PMap Γ → μ (PMap Γ)
PNil ⊕ PNil = return PNil
PCons Nothing s ⊕ PCons Nothing s' = liftM (PCons Nothing) (s ⊕ s')
PCons Nothing s ⊕ PCons (Just y) s' = liftM (PCons (Just y)) (s ⊕ s')
PCons (Just x) s ⊕ PCons Nothing s' = liftM (PCons (Just x)) (s ⊕ s')
PCons (Just x) s ⊕ PCons (Just y) s'
  | x ≡' y = liftM (PCons (Just x)) (s ⊕ s')
  | otherwise = fail "merging failed"
```

Here, `liftM` is the function from Haskell’s standard libraries that lifts a given unary function into an arbitrary monad:

```haskell
liftM :: Monad μ ⇒ (α → β) → μ α → μ β
```

If, for at least one reference, the arguments of the monadic merge operator ⊕ produce different terms, merging fails. As all terms contained in a partial map are of types in the type class Rewritable, equality of terms can be tested by means of the generic equality test ≡'.

### 3.5.2 Generic patterns

Recall from the definition of the datatype `Rule` that the left- and right-hand sides of rewrite rules are represented by values of the type `Pattern Γ α`, where α is the type of terms to be rewritten and Γ is a metavariable environment. The idea is to derive the definition of `Pattern Γ α` from the definition of α, much like in Section 3.2.2 the definition of `EProp` was derived from the definition of `Prop`, but without requiring the user to explicitly declare the pattern type. As pattern
types are supposed to hold the same values as their corresponding term types, but additionally allow each subterm to be replaced by a metavariable. Pattern can be elegantly defined in terms of a so-called type-indexed datatype. A type-indexed datatype (Hinze et al., 2004) is a datatype that is defined by induction over the structure of generically representable types.

Here, we encode type-indexed datatypes as datatype families (Schrijvers et al., 2008). That is, we define a datatype family Extended:

\[
\text{data family } \text{Extended} :: \ast \rightarrow \ast \rightarrow \ast
\]

A type Extended \( \Gamma \alpha \) is to be interpreted as the type that is obtained from extending \( \alpha \) with metavariables from \( \Gamma \).

Instances of Extended are given for all representation constructors. These instances recursively introduce metavariable alternatives in all subterm positions in the generic representation of a type’s structure, while duplicating the remainder of the structure. A pattern is then defined as either a duplicate of a type’s structure with metavariable alternatives for all subterm positions or otherwise a metavariable of the appropriate type:

\[
\text{type } \text{Pattern } \Gamma \alpha = \text{Extended } \Gamma (\text{Rep } \alpha) \rightarrow \text{Ref } \Gamma \alpha
\]

As values of base types do not contain subterms, the extension of these types amounts to mere duplication:

\[
\text{data instance } \text{Extended } \Gamma \text{ Int } = \text{Int}’ \text{ Int}
\]

The cases for Float and Char are analogous; in the sequel, we provide instance declarations for Int as representatives for all base types.

Note that in our library subterm positions in a type are encoded as elements of type-level lists. Hence, sums and lists are extended recursively with metavariable alternatives inserted for all list elements:

\[
\begin{align*}
\text{data instance } \text{Extended } \Gamma \text{ Nil } & = \text{Nil}’ \\
\text{data instance } \text{Extended } \Gamma (\alpha : \ast : \beta) & = \text{Inl}’ (\text{Extended } \Gamma \alpha) \\
& \mid \text{Inr}’ (\text{Extended } \Gamma \beta) \\
\text{data instance } \text{Extended } \Gamma (\alpha :: \beta) & = \text{Pattern } \Gamma \alpha :: \text{’ Extended } \Gamma \beta
\end{align*}
\]

Because extended types contain at least the values of the original representation types (modulo renaming of constructors and redirections into sum types), converting from terms to patterns is straightforward. First, we declare a type class Extensible of types that can be lifted into their extended counterparts:

\[
\text{class Extensible } \alpha \text{ where} \\
\text{extend } :: \alpha \rightarrow \text{Extended } \text{Nil } \alpha
\]

Then, we define a generic extension function extend’ for constructing patterns from terms:

\[
\begin{align*}
\text{extend’ } :: \text{Rewritable } \alpha \Rightarrow \alpha \rightarrow \text{Pattern } \text{Nil } \alpha \\
\text{extend’ } x & = \text{Inl} (\text{extend } (\text{from } x))
\end{align*}
\]
Note that a value of a type `Pattern Nil α`, due to the empty metavariable environment, is guaranteed to not contain any metavariables.

Lifting base types reduces to wrapping values in extension constructors:

```haskell
instance Extensible Int where
    extend = Int'
```

Extension of the empty list involves converting from `Nil` to `Nil'`:

```haskell
instance Extensible Nil where
    extend Nil = Nil'
```

Sums are extended recursively:

```haskell
instance (Extensible α, Extensible β) ⇒ Extensible (α :+: β) where
    extend (Inl x) = Inl' (extend x)
    extend (Inr y) = Inr' (extend y)
```

For `:::`, we require the first type argument to be rewritable, so that subterms can be lifted generically:

```haskell
instance (Rewritable α, Extensible β) ⇒ Extensible (α ::: β) where
    extend (x ::: xs) = extend' x :::' extend xs
```

The conversion from terms to patterns is used in Section 3.6 for the synthesis of rewrite rules from functions over term types.

### 3.5.3 Generic substitutions

Substitutions are just partial maps over a given metavariable environment:

```haskell
type Substitution Γ = PMap Γ
```

Applying a substitution involves traversing a value of an extended type and replacing all metavariable occurrences by subterms drawn from the partial map in order to obtain a term representation:

```haskell
class Substitutable α where
    substitute :: Monad µ ⇒ Substitution Γ → Extended Γ α → µ α
```

As looking up metavariables in partial maps may fail, `substitute` returns its result in a monad `µ`. To apply a substitution to a pattern, we distinguish between values of extended types and metavariables. In the former case, we use `substitute` to yield a representation and then convert this representation to a term by means of `to`. In the latter case, the metavariable is looked up in the partial map that represents the substitution:

```haskell
substitute' :: (Rewritable α, Monad µ) ⇒
    Substitution Γ → Pattern Γ α → µ α

substitute' s (Inl e) = liftM to (substitute s e)
substitute' s (Inr r) = lookup r s
```
Substitutions over extended base types are performed by stripping off the extension constructors:

\[
\text{instance } \text{Substitutable } \text{Int where}
\]
\[
\text{substitute } _\text{Int'}n = \text{return } n
\]

Similarly, for the empty lists of constructor arguments, we define:

\[
\text{instance } \text{Substitutable } \text{Nil where}
\]
\[
\text{substitute } _\text{Nil'} = \text{return } \text{Nil}
\]

Extended sum values are processed recursively and the obtained values are re-injected into the appropriate side of the original sum type:

\[
\text{instance } (\text{Substitutable } \alpha, \text{Substitutable } \beta) \Rightarrow \text{Substitutable } (\alpha : : : \beta) \text{ where}
\]
\[
\text{substitute } s \text{ (Inl' } e) = \text{liftM } \text{Inl } (\text{substitute } s \text{ e})
\]
\[
\text{substitute } s \text{ (Inr' } e) = \text{liftM } \text{Inr } (\text{substitute } s \text{ e})
\]

The instance for : : : once more requires all elements in a list to be in the type class \text{Rewritable} and invokes the generic function \text{substitute'} to apply substitutions to patterns:

\[
\text{instance } (\text{Rewritable } \alpha, \text{Substitutable } \beta) \Rightarrow \text{Substitutable } (\alpha : : : \beta) \text{ where}
\]
\[
\text{substitute } s \text{ (pat } :: : : \text{'} es) =
\]
\[
\text{liftM2 } (:::) \text{ (substitute' } s \text{ pat) (substitute } s \text{ es)}
\]

To lift the list constructor : : : into a monad, this instance uses the standard function \text{liftM2}:

\[
\text{liftM2 } : : : \text{Monad } \mu \Rightarrow (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \mu \alpha \rightarrow \mu \beta \rightarrow \mu \gamma
\]

This function turns binary functions into monadic operations.

### 3.5.4 Generic pattern matching

Finally, let us consider how substitutions are constructed, namely, by generically matching term values against patterns. The required machinery includes a type class \text{Matchable} of representation types which can be matched against their recursively extended counterparts:

\[
\text{class } \text{Matchable } \alpha \text{ where}
\]
\[
\text{match } : : (\text{Mappable } \Gamma, \text{Monad } \mu) \Rightarrow
\]
\[
\text{Extended } \Gamma \alpha \rightarrow \alpha \rightarrow \mu (\text{Substitution } \Gamma)
\]

Also, it includes a top-level generic function \text{match'} for matching terms against either an extended representation or otherwise a top-level metavariable:

\[
\text{match'} : : (\text{Rewritable } \alpha, \text{Mappable } \Gamma, \text{Monad } \mu) \Rightarrow
\]
\[
\text{Pattern } \Gamma \alpha \rightarrow \alpha \rightarrow \mu (\text{Substitution } \Gamma)
\]
\[
\text{match'} \text{ (Inl } e) \ x = \text{match } e \text{ (from } x)
\]
\[
\text{match'} \text{ (Inr } r) \ x = \text{return } (\text{singleton } r \ x)
\]
If a term \( x \) is to be matched against an extended representation \( e \), \( x \) is itself converted to a generic representation from \( x \) and matched by means of \( \text{match} \). If \( x \) is matched against a metavariable \( r \), a singleton substitution is constructed that maps \( r \) to \( x \). Pattern-match failures are dealt with monadically.

Matching values of base type against extended base values requires an equality test. If this test succeeds, an empty substitution is produced; otherwise, a mismatch is reported:

\[
\text{instance } \text{Matchable } \text{Int} \text{ where } \\
\text{match } (\text{Int'} n) n' | n \equiv n' = \text{return empty} \\
| \text{otherwise } = \text{fail "pattern mismatch"}
\]

Provided that both the extended representation and the term representation are completely defined (i.e., do not diverge), matching is always successful for empty lists:

\[
\text{instance } \text{Matchable } \text{Nil} \text{ where } \\
\text{match } \text{Nil} \text{'} \text{Nil } = \text{return empty}
\]

For values of sum types, we check whether the extended representation and the term representation encode the same alternative. If so, we proceed recursively, otherwise, matching fails:

\[
\text{instance } (\text{Matchable } \alpha, \text{Matchable } \beta) \Rightarrow \text{Matchable } (\alpha : : : \beta) \text{ where } \\
\text{match } (\text{Inl' } e) (\text{Inl } x) = \text{match } e x \\
\text{match } (\text{Inr' } e) (\text{Inr } y) = \text{match } e y \\
\text{match } _ - _ = \text{fail "pattern mismatch"}
\]

For nonempty lists, we attempt to match the head \( x \) against a pattern \( \text{pat} \) by means of a call to the generic function \( \text{match' } \) and the tail \( \text{xs} \) against extended representations \( \text{es} \) through a recursive call to \( \text{match} \). If both \( x \) and \( \text{xs} \) are matched successfully, the resulting substitutions are merged with the operator \( \oplus \) defined earlier in Section 3.5.1:

\[
\text{instance } (\text{Rewritable } \alpha, \text{Matchable } \beta) \Rightarrow \text{Matchable } (\alpha :: : : \beta) \text{ where } \\
\text{match } (\text{pat } :: : : t h e s e s ) (x :: :: \text{xs}) = \\
\text{join } (\text{liftM2 } (\oplus) (\text{match' } \text{pat } x) (\text{match es xs}))
\]

As both matching and merging may fail, this gives rise to a nested monadic structure, which we flatten with a call to the function \( \text{join} \) from the standard libraries:

\[
\text{join } : : \text{Monad } \mu \Rightarrow \mu (\mu \alpha) \Rightarrow \mu \alpha
\]

This completes our implementation of generic matching and substitution.

### 3.6 Synthesising rewrite rules

In the previous section, we demonstrated how rewrite rules are extensionally represented using the type synonym \( \text{Pattern} \) and the type-indexed datatype \( \text{Extended} \).
Implementing patterns through generic types frees the user of the library from the burden of defining separate datatypes for representing the left- and right-hand sides of rewrite rules for various term types, but still allows us to enjoy the benefits of observable rules.

However, this use of generic types raises the question how the user is supposed to define rewrite rules. For example, the rewrite rule derived from the principle of contradiction could be written as follows:

\[
\text{contradiction} :: \text{Rule Prop}
\]

\[
\text{contradiction} = \text{Rule lhs rhs}
\]

where

\[
\begin{align*}
\text{lhs} &= \text{Inl} \ (\text{Inr} \ (\text{Inr} \ (\text{Inr} \ (\text{Inl} \ (\text{Inr} \ \text{RZero} :: :: \text{Nil})))))) \\
\text{rhs} &= \text{Inl} \ (\text{Inr} \ (\text{Inl} \ (\text{Inr} \ \text{Nil})))
\end{align*}
\]

But clearly this style of definition is tedious and, moreover, error-prone. Of course, the definition of so-called smart constructors make take away some of the user’s burden:

\[
(\land') :: \text{Pattern } \Gamma \text{ Prop} \to \text{Pattern } \Gamma \text{ Prop} \to \text{Pattern } \Gamma \text{ Prop}
\]

\[
p \land' q = \text{Inl} \ (\text{Inr} \ (\text{Inr} \ (\text{Inl} \ (\text{Inr} \ \text{RZero} :: :: \text{Nil}))))))
\]

However, these smart constructors then need to be defined for all types of rewritable terms, defeating the very purpose of datatype-generic programming. Instead, our library allows for rewrite rules to be defined in terms of the real constructors of the type of terms that are to be rewritten. A library function \text{synthesize} then takes care of translating the terms in rewrite rules into their generic representations. The rule above, for example, can conveniently and concisely be written as follows:

\[
\text{contradiction} :: \text{Rule Prop}
\]

\[
\text{contradiction} = \text{synthesize} \ (\lambda p \to p : \land : \text{Not } p \mapsto F)
\]

That is, rewrite rules are synthesised from functions that take placeholders for metavariables as arguments and produce values of the type of rewritable terms; in this case, Prop. This way, rewrite rules are specified in the same way for different term types, while the internal representation of the rules remains hidden from the user.

To synthesise rules from functions, we develop some more generic machinery. The idea is to instantiate each function parameter twice, each time with distinct term values, and compare the resulting values. This approach restricts us to function parameters of types that have at least two values, but note that this restriction is by no means essential as rules with metavariables that range over types that have only one value are not meaningful.

For the function \(\lambda p \to p : \land : \text{Not } p \mapsto F\) that we used above, we could instantiate the parameter \(p\) first with the value \(T\) and then with the value \(F\). The first instantiation then yields the left-hand side \(T : \land : \text{Not } T\) and the right-hand \(F\);
the second instantiation yields $F : \land: \lnot F$ and $F$. Next, we compare the obtained pairs of left- and right-hand sides to determine where metavariables are to be inserted. As, in our example, the produced left-hand sides differ in the left operand of $\land$: and in the argument of $\lnot$, an occurrence of some metavariable is inserted in these positions. The two right-hand sides are identical, so no metavariable occurrence will show up there.

In this section, we implement this scheme of producing rewrite rules generically. We first show how to generate pairs of distinct values for term types (Section 3.6.1). Then, we present a generic $\text{diff}$ function that localises the positions in which metavariables are to be inserted (Section 3.6.2). Finally, a type class of synthesiser types is given (Section 3.6.3).

### 3.6.1 Generic sampling

To produce pairs of distinct values for types in $\text{Rewritable}$, we define a type class $\text{Sampleable}$:

```
class Sampleable α where
  left :: α
  right :: α
```

Instances of $\text{Sampleable}$ are supposed to have their methods $\text{left}$ and $\text{right}$ produce values that differ in their top-level constructors. With instances of $\text{Sampleable}$ declared for all generic representation types, functions $\text{left}'$ and $\text{right}'$ can be defined generically for all types of rewritable terms:

```
left', right' :: Rewritable α ⇒ α
left' = to left
right' = to right
```

As always, appropriate instances for the base types, such as $\text{Int}$, are straightforward to produce:

```
instance Sampleable Int where
  left = 0
  right = 1
```

For $\text{Nil}$, it is not possible to produce distinct $\text{left}$ and $\text{right}$ values. Still, we have to provide an instance declaration to allow for types that contain $\text{Nil}$ values to be in $\text{Sampleable}$:

```
instance Sampleable Nil where
  left = Nil
  right = Nil
```

As a result, no metavariables are ever introduced in rules over types with only a single nonbottom value. Sum types are easy: here we have the opportunity to
actually produce values that are distinct in their top-level constructor. For \texttt{left}, we choose \texttt{Inl}, while for \texttt{right}, \texttt{Inr} is selected\(^2\):

\begin{verbatim}
instance (Sampleable α, Sampleable β) ⇒ Sampleable (α ::: β) where
  left  = Inl left
  right = Inr left
\end{verbatim}

For ::::, we have only one constructor at our disposal, so a distinction in top-level constructors is to be made at a deeper level:

\begin{verbatim}
instance (Rewritable α, Sampleable β) ⇒ Sampleable (α :::: β) where
  left  = left' :::: left
  right = right' :::: right
\end{verbatim}

### 3.6.2 Generic diff

To determine at which positions in a pattern metavariables are to be introduced, we require the ability to generically compute a ‘diff’ between two patterns. However, it is statically not known at which positions the values differ. Hence, if such a position is found, it depends on the type of that position and the type of the metavariable to be introduced whether or not a new pattern can be distilled. To this end, we use dynamic typing and require term types to be in the type class \texttt{Typeable}, as mentioned in Section 3.3.2. This allows us to compare the types of the positions in which the values differ to the type of the metavariable to be introduced at run time. The type class \texttt{Typeable} comes with a function \texttt{gcast}:

\begin{verbatim}
gcast :: (Typeable α, Typeable β) ⇒ ϕ α → Maybe (ϕ β)
\end{verbatim}

This function casts a value of type \(ϕ \ α\) into a value of type \(ϕ \ β\) if and only if \(α\) and \(β\) are the same type.

First, we define a type class \texttt{Diffable} of representation types for which a diff can be computed:

\begin{verbatim}
class Diffable α where
diff :: Typeable β ⇒
  Extended Γ α → Extended Γ α → Maybe (Extended (β :::: Γ) α)
\end{verbatim}

For each generic representation type \(α\), the overloaded function \texttt{diff} takes two values of type \(Extended \ Γ \ α\) for some environment \(Γ\) and attempts to introduce a new metavariable of type \(β\) at the deeper levels in which the two values differ. If the two values differ at top level or at an inappropriately typed deeper level, \texttt{diff} fails and produces \texttt{Nothing}.

\(^2\)There is a minor caveat associated with the given \texttt{Sampleable}-declaration for sum types. Our library requires the values for \texttt{left} and \texttt{right} to be finite as infinite values will lead to nontermination of the generic diff function in Section 3.6.2. To guarantee termination, we require the leftmost constructor of a datatype to be nonrecursive, such that \texttt{left} always produces a finite value for this constructor. Note that this may require an implicit reordering of constructors when defining or generating generic representations and precludes types that have no finite values. Then, we can use a single \texttt{left}-produced value in the definitions of \texttt{left} and \texttt{right} for \(\ast\) as the top-level constructors \texttt{Inl} and \texttt{Inr} already distinguish the values.
Diffs for rewritable terms are now computed by means of the following generic function \( \text{diff}' \):

\[
\text{diff}' :: (\text{Rewritable } \alpha, \text{Typeable } \beta) \Rightarrow
\begin{align*}
\text{Pattern } \Gamma \alpha & \rightarrow \text{Pattern } \Gamma \alpha \rightarrow \text{Maybe} (\text{Pattern } (\beta :::: \Gamma) \alpha) \\
\text{diff}' (\text{Inl } e) (\text{Inl } e') &= \text{case } \text{diff } e e' \text{ of} \\
& \quad \text{Nothing } \rightarrow \text{gcast } (\text{Inr } \text{RZero}) \\
& \quad \text{Just } e'' \rightarrow \text{Just } (\text{Inl } e'') \\
\text{diff}' (\text{Inr } r) (\text{Inr } r') & | r \equiv r' = \text{Just } (\text{Inr } (\text{RSucc } r)) \\
\text{diff}' - - &= \text{Nothing}
\end{align*}
\]

This generic function takes patterns over a type \( \alpha \) as argument and introduces a new metavariable of type \( \beta \). If both patterns are metavariable alternatives \( \text{Inr } r \) and \( \text{Inr } r' \), we require \( r \) and \( r' \) to be the same metavariable and increment the corresponding metavariable. Consequently, \( \text{RZero} \) becomes a fresh metavariable again that \( \text{diff}' \) can safely introduce. If both patterns consist of values \( e \) and \( e' \) of an extended type, the overloaded \( \text{diff} \) function is used to compare these values. When \( \text{diff} \) fails, we have found a position where the fresh metavariable \( \text{RZero} \) must be introduced. Insertion of such a metavariable is only allowed if the type \( \alpha \) of this position and the type \( \beta \) of the metavariable are the same, as also dictated by the type of \( \text{RZero} \). We use the function \( \text{gcast} \) to compare \( \alpha \) and \( \beta \) at run time and cast the value of type \( \text{Pattern} (\beta :::: \Gamma) \beta \) into a value of type \( \text{Pattern} (\beta :::: \Gamma) \alpha \).

When \( \text{diff} \) successfully computes a combined value \( e'' \), this value is wrapped in a pattern \( \text{Inl } e'' \) and returned.

It remains to give instances of \( \text{Diffable} \) for our generic representation constructors. As values of base types contain no subterms and can thus only differ at top level, an implementation of \( \text{diff} \) for these types reduces to testing for equality:

\[
\text{instance } \text{Diffable } \text{Int} \text{ where} \\
\text{diff } (\text{Int} n) (\text{Int} n') | n \equiv n' = \text{Just } (\text{Int} n) \\
| \text{otherwise } = \text{Nothing}
\]

The extension of \( \text{Nil} \) holds only a single value, so \( \text{diff} \) for empty lists cannot fail:

\[
\text{instance } \text{Diffable } \text{Nil} \text{ where} \\
\text{diff } \text{Nil} \text{ Nil' } = \text{Just } \text{Nil'}
\]

For values of sum type, we compare the top-level constructors. If these are different, we produce \( \text{Nothing} \); otherwise, comparison proceeds recursively:

\[
\text{instance } (\text{Diffable } \alpha, \text{Diffable } \beta) \Rightarrow \text{Diffable } (\alpha :::: \beta) \text{ where} \\
\text{diff } (\text{Inl' } e) (\text{Inl' } e') = \text{liftM } \text{Inl' } (\text{diff } e e') \\
\text{diff } (\text{Inr' } e) (\text{Inr' } e') = \text{liftM } \text{Inr' } (\text{diff } e e') \\
\text{diff } - - = \text{Nothing}
\]

Similarly, for \( :::: \), the comparison of two values \( \text{pat} :::: \text{es} \) and \( \text{pat}' :::: \text{es}' \) continues recursively underneath the constructor \( ::::' \):

\[
\text{instance } (\text{Rewritable } \alpha, \text{Diffable } \beta) \Rightarrow \text{Diffable } (\alpha :::: \beta) \text{ where} \\
\text{diff } (\text{pat} :::: \text{es}) (\text{pat'} :::: \text{es'}) = \\
\text{liftM2 } (::::') (\text{diff'} \text{pat pat'}) (\text{diff es es'})
\]

63
3.6.3 Generic synthesis

With generic sampling and generic diff defined, we can now implement the synthesis of rewrite rules from functions over term types. These functions wrap the left- and right-hand sides of rules in values of a type \( \text{Template} \) of which the values simply constitute pairs of terms:

\[
\text{data } \text{Template } \alpha = \text{Template } \alpha \alpha
\]

For the concise definition of templates, we introduce an operator \( \mapsto \mapsto \):

\[
infix 1 \mapsto \mapsto \quad (\mapsto \mapsto) :: \alpha \rightarrow \alpha \rightarrow \text{Template } \alpha
\]

\[
\text{lhs } \mapsto \text{rhs } = \text{Template } \text{lhs } \text{rhs}
\]

Next, we define a type class \( \text{Synthesiser} \) of types of which the values can be used to synthesise rewrite rules:

\[
\text{class } \text{Rewritable } (\text{Term } \alpha) \Rightarrow \text{Synthesiser } \alpha \text{ where}
\]

\[
\quad \text{type } \text{Term } \alpha :: \star
\]

\[
\quad \text{type } \text{Env } \alpha :: \star
\]

\[
\quad \text{patterns } :: \alpha \rightarrow (\text{Pattern } (\text{Env } \alpha) (\text{Term } \alpha), \text{Pattern } (\text{Env } \alpha) (\text{Term } \alpha))
\]

Each instance \( \alpha \) of \( \text{Synthesiser} \) has an associated type synonym \( \text{Term } \alpha \) that gives the type of terms that are rewritten by a synthesised rewrite rule. Similarly, the associated type synonym \( \text{Env } \alpha \) gives the term types over which the metavariables of a synthesised rule range. For example, a rewrite rule synthesised from a function of a type \( \alpha \rightarrow \beta \rightarrow \text{Template } \gamma \) has two metavariables, ranging over values of types \( \alpha \) and \( \beta \), and is used to rewrite terms of type \( \gamma \). Operationally, a value \( x \) of a type from \( \text{Synthesiser} \) can be used to produce a pair \( \text{patterns } x \) that contains the left- and right-hand-side components of a rewrite rule. Synthesis then reduces to combining these components in a \( \text{Rule} \) value:

\[
\text{synthesise } :: (\text{Synthesiser } \alpha, \text{Mappable } (\text{Env } \alpha)) \Rightarrow \alpha \rightarrow \text{Rule } (\text{Term } \alpha)
\]

\[
\text{synthesise } x = \text{let } (\text{lhs }, \text{rhs }) = \text{patterns } x
\]

\[
\text{in } \text{Rule } \text{lhs } \text{rhs}
\]

Instances of the type class \( \text{Synthesiser} \) are defined inductively over the structure of function types. As a base case, we have an instance for \( \text{Template } \alpha \) for any type \( \alpha \) of rewritable terms:

\[
\text{instance } \text{Rewritable } \alpha \Rightarrow \text{Synthesiser } (\text{Template } \alpha) \text{ where}
\]

\[
\text{type } \text{Term } (\text{Template } \alpha) = \alpha
\]

\[
\text{type } \text{Env } (\text{Template } \alpha) = \text{Nil}
\]

\[
\text{patterns } (\text{Template } \text{lhs } \text{rhs }) = (\text{extend'} \text{lhs }, \text{extend'} \text{rhs })
\]

Rewrite rules that are synthesised directly from templates over \( \alpha \) operate on terms of type \( \alpha \) and contain no metavariables. Left- and right-hand sides for these rules can be obtained simply by lifting template components into the type \( \text{Pattern } \text{Nil } \alpha \)
of patterns over $\alpha$ without variables, for which we use the generic function $\text{extend}'$ defined in Section 3.5.2.

In the inductive step, we require, in order for a function type $\alpha \rightarrow \beta$ to be in the type class $\text{Synthesiser}$, $\alpha$ to be a type of rewritable terms and $\beta$ to be a type of synthesisers:

$$\text{instance} \ (\text{Rewritable } \alpha, \text{Synthesiser } \beta) \Rightarrow \text{Synthesiser} \ (\alpha \rightarrow \beta) \ \text{where}$$

- $\text{type} \ \text{Term} \ (\alpha \rightarrow \beta) = \text{Term} \ \beta$
- $\text{type} \ \text{Env} \ (\alpha \rightarrow \beta) = \alpha :: :: \text{Env} \ \beta$
- $\text{patterns} \ f = \text{let} \ (\text{lhs}, \text{rhs}) = \text{patterns} \ (f \ \text{left}')$
  \hspace{1cm} $\text{in} \ \text{case} \ (\text{diff}' \ \text{lhs} \ \text{lhs'}, \ \text{diff}' \ \text{rhs} \ \text{rhs'}) \ \text{of}$
  \hspace{1cm} $(\text{Just} \ \text{lhs}'', \ \text{Just} \ \text{rhs}'') \Rightarrow (\text{lhs}'', \ \text{rhs}'')$
  \hspace{1cm} $\rightarrow \ \text{error} \ "\text{synthesis failure}"$

Function abstraction over $\alpha$ adds one $\alpha$-typed metavariable to the environment $\text{Env} \ \beta$, but does not alter the type $\text{Term} \ \beta$ of terms the synthesised rule operates on. Patterns of the left- and right-hand sides of the rewrite rule are constructed by applying the function twice (once to the value produced by $\text{left}'$ and once to the value produced by $\text{right}'$) and then computing diffs from the obtained components, possibly introducing occurrences of a new metavariable that ranges over terms of type $\alpha$.

As an example of how this machinery works, consider a rewrite rule $f$ that uses two metavariables. First, the patterns for $f \ \text{left}'$ are constructed, which in its turn constructs patterns for $f \ \text{left}' \ \text{left}'$ and $f \ \text{left}' \ \text{right}'$, which again are lifted into actual patterns using $\text{extend}'$. A diff is computed for the left- and right-hand sides of these actual patterns, resulting in an instantiation of $f$ where the first metavariable remains fixed to $\text{left}'$ and the second metavariable becomes $\text{RZero}$ as the values at that position differ. Second, the patterns for $f \ \text{right}'$ are constructed in a similar fashion, now resulting in an instantiation of $f$ where the first metavariable remains fixed to $\text{right}'$ and the second metavariable is $\text{RZero}$ again. Finally, a diff is computed for the two patterns with the second metavariable already inserted, resulting in an instantiation of $f$ where also the first metavariable is inserted as $\text{RZero}$ and the second metavariable is incremented to $\text{RSucc} \ \text{RZero}$.

Note that our use of $\text{diff}'$ cannot fail. Its two arguments are instantiations of the same rewrite rule, where the metavariable positions contain either extended values described by $\text{left}'$ and $\text{right}'$, or equal metavariables that were previously inserted. Also, the use of $\text{gcast}$ in $\text{diff}'$ is safe: we only differ the value of one metavariable at a time and we guarantee that the type of the corresponding positions is the same as the type of the metavariable to be introduced, as captured by the associated type synonym $\text{Env}$.

### 3.7 Detecting ill-formed rewrite rules

In the previous sections, we have shown the implementation of our library’s core functionality. In particular, we have shown how, although we use an extensional representation of rewrite rules internally, we allow the user to define rules in terms
of functions over domain-specific types. Due to this sugarcoating, additional verification of rewrite rules is required.

Consider, for example, the following rewrite rule over propositional formulae:

\[
\text{funny} :: \text{Rule Prop} \\
\text{funny} = \text{synthesise} (\lambda n \to f \ n \mapsto T)
\]

Here, \(f\) is some function taking \(\text{Int}\) values to values of type \(\text{Prop}\):

\[
f :: \text{Int} \to \text{Prop}
\]

It is unclear what the semantics of such a rewrite rule should be. That is, in a well-formed rewrite rule we expect metavariables to exclusively occur as constructor arguments, not as arguments to arbitrary functions. Using Haskell’s variables as placeholders for our metavariables means, however, that we cannot preclude such ill-formed rules and we have to rely on the user not to construct nonsensical rules as the one above.

Another class of meaningless rewrite rules can be excluded by equipping our library with functionality for detecting their ill-formedness. Consider, for instance, the following rule:

\[
\text{unbound} :: \text{Rule Prop} \\
\text{unbound} = \text{synthesise} (\lambda p \to T \mapsto T : \lor : p)
\]

In this rule, the metavariable \(p\) on the right-hand side is not bound on the left-hand side of the rewrite rule. But also, take the following rule:

\[
\text{superfluous} :: \text{Rule Prop} \\
\text{superfluous} = \text{synthesise} (\lambda p \ q \to p : \lor : p \mapsto p)
\]

Here, the metavariable \(q\) is superfluous since it is ‘declared’ but not used at all in the rewrite rule. In general, we consider a rewrite rule well-formed if and only if all of its declared metavariables are bound in its left-hand side. Interestingly, this notion of well-formedness can be checked for statically, i.e., without applying the rule.

To this end, we extend the library with a function \(\text{validate}\) that provides the user with an opportunity to verify the use of declared metavariables in rewrite rules:

\[
\text{validate} :: \text{Rewritable } \alpha \Rightarrow \text{Rule } \alpha \to \text{Bool}
\]

This function is intended to be applied just after rule synthesis.

Validation is achieved by constructing a use record with a field for each metavariable, denoting its presence in the left-hand side of the rewrite rule:

\[
data \text{Record} :: * \to * \text{ where} \\
R\text{Nil} :: \text{Record } \text{Nil} \\
R\text{Cons} :: \text{Bool} \to \text{Record } \Gamma \to \text{Record } (\alpha :: : \Gamma)
\]

An initial blank record is created by setting each presence to \(\text{False}\):
3.7 Detecting ill-formed rewrite rules

```haskell
class Recordable Γ where
  blank :: Record Γ

instance Recordable Nil where
  blank = RNil

instance Recordable Γ ⇒ Recordable (α ::: Γ) where
  blank = RCons False blank
```

We now require environments to be instances of the type class `Recordable` and, hence, a constraint is added to the constructor `Rule` from Section 3.5:

```haskell
data Rule :: * → * where
  Rule :: (⋯, Recordable Γ) ⇒ Pattern Γ α → Pattern Γ α → Rule α
```

A use record is updated by traversing the left-hand side of a rewrite rule and checking off each metavariable encountered:

```haskell
class Validateable α where
  record :: Extended Γ α → Record Γ → Record Γ

instance Validateable Int where
  record (Int _′) = id

instance Validateable Nil where
  record Nil′ = id

Values of sum types are traversed by stripping their top-level constructor:

```haskell
instance (Validateable α, Validateable β) ⇒ Validateable (α :+: β) where
  record (Inl′ e) = record e
  record (Inr′ e) = record e
```

For `:::`, we update the record by traversing the subterms and the pattern in sequence:

```haskell
instance (Rewritable α, Validateable β) ⇒ Validateable (α ::: β) where
  record (pat :::′ es) = record′ pat ∘ record es
```
Note that since the record is only used to check off metavariable use, the order of the calls to record' and record plays no role.

Next, we add a superclass constraint for Validateable to the declaration of the type class Rewritable from Section 3.3.2:

\[
\text{class } (\cdots, \text{Validateable} \ (\text{Rep } \alpha)) \Rightarrow \text{Rewritable } \alpha
\]

And, we define a top-level function for validating rules:

\[
\text{validate} :: \text{Rewritable } \alpha \Rightarrow \text{Rule } \alpha \rightarrow \text{Bool}
\]

\[
\text{validate} \ (\text{Rule lhs } _) = \text{check} \ (\text{record'} \ l.hs \ blank)
\]

\[
\text{where}
\]

\[
\text{check} \ \text{RNil} = \text{True}
\]

\[
\text{check} \ (\text{RCons } b \ \text{rec}) = b \land \text{check} \ \text{rec}
\]

Starting with a blank record, validate records all occurrences of metavariables on the left-hand side of a rewrite rule and then verifies that all metavariables in the environment of the rule are checked off in the updated record.

### 3.8 Guarded rewriting

In the previous section, we have added some infrastructure for statically validating rewrite rules to the core functionality of our library. In this section, we further extend the library and add support for rewrite rules guarded by preconditions.

As an example, consider the following datatype Lam of λ-terms:

\[
\text{data Lam} = \text{Var String} \mid \text{Abs String Lam} \mid \text{App Lam Lam}
\]

Also, assume there is an accompanying function fv that produces the variables that appear free in a given λ-term:

\[
fv :: \text{Lam} \rightarrow \text{[String]}
\]

Now suppose that we want to define a rewrite rule that implements eta-reduction, rewriting \(\lambda x \rightarrow e\ x\) to \(e\) but only if such a term additionally fulfills the precondition that the variable \(x\) does not appear free in the term \(e\). Using the extension presented in this section, such rewrite rules can be written as in:

\[
\text{etaReduction} :: \text{Rule Lam}
\]

\[
\text{etaReduction} = \text{synthesise} \ (\lambda x \ e \rightarrow \text{Abs } x \ (\text{App } e \ (\text{Var } x)) \mapsto e \odot x \notin fv \ e)
\]

Here, we synthesise a rule over λ-terms from a function that produces a template constructed with the operators \(\mapsto\) and \(\odot\). The latter adds a guard to the rewrite rule, i.e., a Boolean expression that may refer to the metavariables abstracted over by the synthesiser function.

In order to implement preconditions, we extend our type Rule of rewrite rules with a component containing a guard:

\[
\text{data Rule} :: \star \rightarrow \star \text{ where}
\]

\[
\text{Rule} :: (\text{Mappable } \Gamma, \text{Recordable } \Gamma, \text{Testable } \Gamma) \Rightarrow
\]

\[
\text{Pattern } \Gamma \alpha \rightarrow \text{Pattern } \Gamma \alpha \rightarrow \text{Guard } \Gamma \rightarrow \text{Rule } \alpha
\]
In addition to the type classes `Mappable` from Section 3.5 and `Recordable` from Section 3.7, metavariable environments used within rules are restricted to be instances of the type class `Testable`, to be explained below. Guard types are defined inductively over the structure of metavariable environments. That is, we have a type family `Guard` and two instances:

\[
\text{type family } \text{Guard} \Gamma :: \ast \\
\text{type instance } \text{Guard} \text{ Nil } = \text{Bool} \\
\text{type instance } \text{Guard} (\alpha :: : : \Gamma) = \alpha \rightarrow \text{Guard} \Gamma
\]

A guard for a rewrite rule without metavariables is just a Boolean expression. For rules that do have metavariables, a guard is a function that takes an argument of appropriate type for each metavariable and produces a Boolean value.

Given a substitution for a metavariable environment `\Gamma` as defined in Section 3.5.3, values of type `Guard \Gamma` can be tested in order to obtain a Boolean that indicates whether the corresponding precondition is fulfilled. To this end, we define the type class `Testable` of environments for which guards are testable:

\[
\text{class } \text{Testable} \Gamma \text{ where} \\
\quad \text{test } :: \text{Guard} \Gamma \rightarrow \text{Substitution} \Gamma \rightarrow \text{Bool}
\]

For the empty-environment type `Nil`, the guard is itself already a value of type `Bool`, so testing can just discard the supplied substitution (which can only be constructed by `PNil` anyway):

\[
\text{instance } \text{Testable} \text{ Nil where} \\
\quad \text{test } b \text{ PNil } = b
\]

For an environment `\alpha :: : : \Gamma`, the guard function is applied to the value that is to be substituted for the metavariable corresponding to `\alpha` and the resulting guard for `\Gamma` is tested recursively:

\[
\text{instance } \text{Testable} \Gamma \Rightarrow \text{Testable} (\alpha :: : : \Gamma) \text{ where} \\
\quad \text{test } f (\text{PCons } \text{(Just } x \text{)} s) = \text{test } s (f x) \\
\quad \text{test } \text{(PCons Nothing )} = \text{error } "\text{test failure}"
\]

If no substitution value is available, the governing rewrite rule was ill-formed due to an unbound metavariable, as described in Section 3.7, and testing fails.

As the datatype `Rule` now requires all metavariable environments to be testable, enforcing preconditions is straightforward:

\[
\text{rewriteWithM } :: (\text{Rewritable } \alpha, \text{Monad } \mu) \Rightarrow \text{Rule} \alpha \rightarrow \alpha \rightarrow \mu \alpha \\
\text{rewriteWithM } (\text{Rule } lhs \text{ rhs } \text{grd}) x = \\
\quad \text{do } s \leftarrow \text{match'} lhs x \\
\quad \quad \text{if test } \text{grd } s \text{ then substitute'} s \text{ rhs } \text{else fail } "\text{precondition failed}"
\]

If, for a given rule `Rule lhs rhs grd` and term `x`, `x` successfully matches against the left-hand side `lhs`, the resulting substitution `s` is tested against the guard `grd`. If the test succeeds, the substitution `s` and the right-hand side `rhs` are combined to produce a new term; otherwise, the rule does not apply and rewriting fails.
What remains is to adapt the synthesis of rules from templates and functions producing templates, as described in Section 3.6. Firstly, we extend templates with a Boolean component:

```
data Template α = Template α α Bool
```

Next, we redefine and introduce the smart constructors \(\mapsto\mid\) and \(\mathbin{/\text{fatsemi}}\), respectively:

```
infix 1 \mapsto
infix 0 \mathbin{/\text{fatsemi}}

\(\mapsto\) :: α → α → Template α
lhs \mapsto rhs = Template lhs rhs True

\(\mathbin{/\text{fatsemi}}\) :: Template α → Bool → Template α
Template lhs rhs \mathbin{/\text{fatsemi}} b = Template lhs rhs b
```

The type class `Synthesiser` now gets an additional method `guard` that produces, for a synthesised rule, a guard of appropriate type:

```
class ··· ⇒ Synthesiser α where

::
guard :: α → Guard (Env α)
```

For rules synthesised directly from templates, this guard is just the Boolean from the template:

```
instance ··· ⇒ Synthesiser (Template α) where

::
guard (Template lhs rhs b) = b
```

For rules synthesised from functions, the guard is itself a function too:

```
instance ··· ⇒ Synthesiser (α → β) where

::
guard f = guard ∘ f
```

The function `synthesise`, finally, that turns synthesisers into rewrite rules simply puts guards in the right places in rules:

```
synthesise :: (Synthesiser α, Mappable (Env α), Testable (Env α)) ⇒ α → Rule (Term α)
synthesise x = let (lhs, rhs) = patterns x
                in Rule lhs rhs (guard x)
```

Note that the given implementation of guarded rewrite rules has one obvious drawback: preconditions are encoded intensionally rather than extensionally and are therefore not observable. This reintroduces some of the problems mentioned in Section 3.2. Most prominently, when pretty-printing rewrite rules, the rendering of preconditions will pose a problem. The other issues listed in Section 3.2 are,
3.9 A case study: solving arithmetic equations

The previous section completed our exploration of our generic library for term rewriting. In this and the next section, we evaluate our approach. In this section, we present (part of) a small case study of a more or less realistic use of our library: solving arithmetic equations using term rewriting. In this case study, we use some of the more advanced features of our library, such as heterogeneously typed metavariables and guarded rewrite rules.

Consider the problem of solving the following equation:

\[
1 + \frac{8}{(x - 3)^2} = 3
\]

To solve such an equation with a single variable, we use the so-called cover-up method, which is based on covering up the part of the equation that contains the variable. We can define cover-up rewrite rules for addition, subtraction, multiplication, division, and exponentiation operations; with these rules we solve the example equation in the following sequence of steps:

\[
\begin{align*}
1 + \frac{8}{(x - 3)^2} &= 3 \\
\Leftrightarrow \quad \frac{8}{(x - 3)^2} &= 2 \\
\Leftrightarrow \quad (x - 3)^2 &= 4 \\
\Leftrightarrow \quad x - 3 &= 2 \quad \lor \quad x - 3 &= -2 \\
\Leftrightarrow \quad x &= 5 \quad \lor \quad x &= 1
\end{align*}
\]

The domain of interest is represented by three components. The first is a variation of the datatype \( \text{Prop} \) from Section 3.1, that allows for formulae to be expressed over atoms of different types:

\[
data \text{Prop} \alpha = \text{Var} \alpha \mid T \mid F \\
\mid \text{Not} (\text{Prop} \alpha) \\
\mid \text{Prop} \alpha : \land: \text{Prop} \alpha \mid \text{Prop} \alpha : \lor: \text{Prop} \alpha
\]

The second is a type for equations:

\[
data \text{Equation} \alpha = \alpha : \equiv: \alpha
\]

And the third a type \( \text{Expr} \) of various arithmetic expressions:

\[
data \text{Expr} = \text{Const} \text{ Rational} \mid \text{Varia String} \\
\mid \text{Expr} : \ast: \text{Expr} \mid \text{Expr} : :\text{Expr} \\
\mid \text{Expr} : \ast: \text{Expr} \mid \text{Expr} : /: \text{Expr} \\
\mid \text{Expr} : :\text{Expr}
\]
For each of these datatypes we need instances of the type classes `Representable` (as described in Section 3.3) and of course `Rewritable` (a single line).

Using the three datatypes, the equation \(1 + \frac{8}{(x-3)^2} = 3\) can be represented as follows:

\[
\text{Var} \left( ((\text{Const} \ 1 :+: (\text{Const} \ 8 :/: ((\text{Var} \ "x" :-> \ Const \ 3) :-> \ Const)) :==: \ Const \ 3)) \right)
\]

The solution to this equation, \(x = 5 \lor x = -1\), is represented as:

\[
\text{Var} \ (\text{Var} \ "x" :==: \ Const \ 5) :\lor: \ \text{Var} \ (\text{Var} \ "x" :==: \ Const \ (-1))
\]

Our rewrite system consists of simple rules for simplifying propositions, such as the following:

\[
\text{orTrueLeft} :: \text{Rewritable} \ \alpha \Rightarrow \text{Rule} (\text{Prop} \ \alpha)
\]
\[
\text{orTrueLeft} = \text{synthesise} (\lambda p \to T :\lor: p \mapsto p)
\]

But also some rules for rewriting additions, which require preconditions:

\[
\text{coverPlusLeft} :: \text{Rule} (\text{Equation} \ \text{Expr})
\]
\[
\text{coverPlusLeft} = \text{synthesise} (\lambda x \ y \ z \to x :+: y :==: z \mapsto x :==: z :-> y \ ; \ \text{hasVaria} \ x \land \text{noVaria} \ y)
\]

In the rule `coverPlusLeft`, all metavariables range over expressions. We only want to apply this rule if there are variables in the expression \(x\) and no variables in the expression \(y\), so as to guarantee the isolation of the variables on the left-hand side of the equation. The helper functions `hasVaria` and `noVaria` test the presence and absence of variables in an expression.

Dealing with exponentiation requires a more complex rule:

\[
\text{coverPowerEven} :: \text{Rule} (\text{Prop} (\text{Equation} \ \text{Expr}))
\]
\[
\text{coverPowerEven} = \text{synthesise} (\lambda x \ n \ y \to
\begin{align*}
\text{let} & \ z = y :-> \ Const \ (1 / n) \\
\text{in} & \ \text{Var} \ (x :-> \ Const \ n :==: y) \mapsto \\
& \ \text{Var} \ (x :==: z) :\lor: \ \text{Var} \ (x :==: \ Const \ 0 :->: z) \ ; \ \text{hasVaria} \ x \land n > 0 \land \text{isEven} \ n)
\end{align*}
\]

As this definition illustrates, complex rewrite rules can be become quite verbose, but we can freely use local definitions to keep rules more or less readable. Since our rewrite rules are observable, a pretty-printer would be able to format such rules nicely. Note, however, that guards in rewrite rules are not observable since these are just Boolean values, as described earlier in Section 3.8.

### 3.10 Performance

The biggest disadvantage of generic programming techniques is that they can be a source of inefficiency. The introduction of representation types and corresponding conversions to and from the original datatypes generally imposes a penalty
on execution time. We have measured the performance of our generic rewriting
library to assess how large this penalty is, compared to hand-written code for a
specific datatype. We have performed two separate tests of different complexities.
The first one deals with logical propositions and uses neither preconditions nor
metavariables of different types. The second one deals with arithmetic equations,
and uses the full power of our generic rewriting library. Both are bundled with
the library for analysis and repeatability.

### 3.10.1 Turning propositions into disjunctive normal form

Our first benchmark uses the datatype $Prop$ of propositional formulae from Sec-
tion 3.1, extended with constructors for implication and equivalence. We have
defined 16 rewrite rules and used these rules to bring the logical proposition to
disjunctive normal form (DNF). This rewrite system is a realistic application of
our rewriting library, and is very similar to the system that is used in an exer-
cise assistant for e-learning systems (Heeren et al., 2008). None of the rules have
preconditions, and all metavariables are of type $Prop$.

Conversion to DNF has been tested with four different strategies: such a
strategy controls which rewrite rule is tried, and where. The strategies range
from naive (i.e., apply some rule somewhere), to more involved strategy specific-
ations that stage the rewriting and use all kinds of traversal combinators. We
implemented these combinators in a type-specific fashion. They could also be im-
plemented as generic functions, and not necessarily with the library we present.
However, this would add another source of inefficiency to our tests, one that we
do not wish to benchmark, hence our choice for implementing the strategies in a
type-specific fashion.

We use QuickCheck (Claessen and Hughes, 2000) to generate a sequence of
random propositions. The random-number generator is initiated with a fixed seed
so that the same sequence is used for all test runs. We carefully profiled our
tests to assure that the computation time was being spent mostly on the rewriting
functionality, and not on auxiliary infrastructure such as data generation.

Because the strategy highly influences how many rules are actually tried, we
vary the number of terms that has to be brought to disjunctive normal form
depending on the strategy that is used. Table 3.1 shows for each strategy the
number of terms that are normalised, how many rules are successfully applied,
and the total number of rules that have been fired. The final column shows the
percentage of rules that succeeded: the numbers reflect that the simpler strategies
fire more rules.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Terms</th>
<th>Rules applied</th>
<th>Rules tried</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>dnf-1</td>
<td>10,000</td>
<td>217,076</td>
<td>113,511,244</td>
<td>0.19%</td>
</tr>
<tr>
<td>dnf-2</td>
<td>50,000</td>
<td>492,114</td>
<td>22,224,222</td>
<td>2.21%</td>
</tr>
<tr>
<td>dnf-3</td>
<td>50,000</td>
<td>487,490</td>
<td>22,467,730</td>
<td>2.17%</td>
</tr>
<tr>
<td>dnf-4</td>
<td>100,000</td>
<td>872,494</td>
<td>18,327,913</td>
<td>4.76%</td>
</tr>
</tbody>
</table>

Table 3.1: The strategies benchmarked
A lightweight approach to datatype-generic rewriting

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Absolute (s)</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PM</td>
<td>SR</td>
</tr>
<tr>
<td>dnf-1</td>
<td>3.11</td>
<td>10.89</td>
</tr>
<tr>
<td>dnf-2</td>
<td>2.52</td>
<td>4.82</td>
</tr>
<tr>
<td>dnf-3</td>
<td>2.49</td>
<td>4.87</td>
</tr>
<tr>
<td>dnf-4</td>
<td>3.94</td>
<td>7.28</td>
</tr>
</tbody>
</table>

Table 3.2: Benchmark results for the Prop datatype with -O1

We compare the execution times of three different implementations for the collection of rewrite rules:

**Pattern Matching (PM)** The first implementation defines the 16 rewrite rules as functions that use pattern matching. This implementation suffers from all the drawbacks that were mentioned in Section 3.1, making this version less suitable for an actual application. However, this implementation of the rules is worthwhile to study because Haskell has excellent support for pattern matching, which will likely result in efficient code.

**Specialised Rewriting (SR)** We have also implemented a specialised rewriting system that operates on propositions, very much like that described in Section 3.2.2. The most significant difference is that we have reused the Var constructor for representing metavariables too, thus mixing object variables with metavariables and avoiding the need to introduce an additional, extended datatype of propositions.

**Generic Rewriting (GR)** Here, we implemented the rules using the generic functions for rewriting that are introduced in this chapter. The instance of the Representable type class is similar to the declaration in Section 3.3, except that it also includes the constructors for equivalence and implication.

All test runs were executed on a machine running Windows XP Professional x64 Edition with SP2 on an Intel Core 2 Duo 3Ghz with 2GB of RAM. The programs were compiled with GHC 6.10.4 with standard optimisation level (using the -O1 compiler flag). We do not use optimisation level -O2 because we noticed that it sometimes reduced performance. Execution times were measured as the difference of the value returned by the function System.CPUTime.getCPUTime from the base libraries that ship with the GHC, after and before the execution of the test, and averaged over 10 runs.

Table 3.2 shows the performance for each implementation of the strategies. The absolute figures are given in seconds, and we also show the figures relative to the pattern-matching approach (PM). The table shows that PM is significantly faster than the other approaches. The specialised rewriting approach (SR) adds observability of the rewrite rules, at the cost of approximately doubling execution time. The generic approach (GR), when compared to the SR approach, suffers from a slowdown of a factor of about 3. This is probably due to the conversions to and from the structure representation of propositions. We also observe a correlation between strategy ratio of rule application as described in Table 3.1 and...
performance: the higher the ratio, the better the performance. This confirms that the overhead of both the SR and GR approaches is caused by the rewriting infrastructure: the PM approach has little overhead from trying rules as it uses Haskell’s native support for pattern matching.

Inspired by Magalhães et al. (2010), we repeated our benchmark by setting the compilation flag \texttt{-funfolding-creation-threshold} to 450 and the compilation flag \texttt{-funfolding-use-threshold} to 60. These flags control, respectively, the keenness of the compiler to export function definitions into interface files and to inline them. This has been shown to increase the performance of certain generic functions, since inlining ‘large’ functions such as \texttt{to} and \texttt{from} exposes opportunities for further optimisations. We show the new results in Table 3.3. Note that the relative figures are still in relation to PM compiled with \texttt{-O1}, as this is the ‘standard’ approach at the ‘standard’ optimisation level. Increased inlining effectively improves the performance. All the approaches benefit from it, but the most pronounced gains are seen in the GR approach, where performance is improved to between 40% and 60% of the original levels. Strategy dnf-4, in particular, shows the highest improvement, now taking only twice as much as the original PM approach.

### 3.10.2 Solving arithmetic equations

Our second benchmark is performed on a family of datatypes representing arithmetic equations, as introduced in Section 3.9. We use 25 rules, some with preconditions and some using metavariables of different types, therefore testing the full potential of our library in a realistic setting. These rules are applied to isolate variables on the left-hand sides of equations.

Again, we have used QuickCheck for test data generation. We test a single strategy, and use type-specific traversals for its application. We compare our library against a pattern-matching approach (PM) only, and again include figures with standard \texttt{-O1} optimisation and with increased inlining as described previously. The results, as an average over 10 runs, are summarised in Table 3.4. We can conclude that the introduction of preconditions and metavariables of different types does not significantly influence performance. Promoting inlining continues to prove useful to increase the performance of our library.

Our benchmarks confirm that observability of rules comes at the expense of loss in run-time efficiency. Furthermore, generic definitions introduce some additional overhead. The trade-off between efficiency and genericity depends on the applic-
Table 3.4: Benchmark results for solving arithmetic equations

<table>
<thead>
<tr>
<th>Optimisation</th>
<th>Absolute (s)</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PM</td>
<td>GR</td>
</tr>
<tr>
<td>Standard</td>
<td>0.57</td>
<td>2.44</td>
</tr>
<tr>
<td>Increased inlining</td>
<td>0.60</td>
<td>1.87</td>
</tr>
</tbody>
</table>

3.11 Related work

Jansson and Jeuring (2000) implement a generic rewriting library in PolyP (Jansson and Jeuring, 1997), an extension of Haskell with a special construct for generic programming. Our library differs in a number of aspects. First, we use no extensions of Haskell specific to generic programming. This is a minor improvement, since we expect that Jansson and Jeuring’s library can easily be translated to plain Haskell as well. Second, we use a type-indexed datatype for specifying rules. This is a major difference, since it allows us to generically extend a datatype with metavariables. In Jansson and Jeuring’s library, a datatype either has to be extended by hand, forcing users to introduce a new constructor, or one of the constructors of the original datatype is to be reused for metavariables. Neither solution is very satisfying, since either functions unrelated to rewriting must now handle the new metavariable constructor, or we are forced to introduce a safety problem in the library since an object variable may accidentally be considered a metavariable.

Libraries that provide us with generic traversal combinators, such as Strafunski (Lämmel and Visser, 2002), Scrap Your Boilerplate (Lämmel and Peyton Jones, 2003), Uniplate (Mitchell and Runcciman, 2007), Bringert’s ‘almost compositional’ functions (Bringert and Ranta, 2006), and probably more, can be used to define intensionally represented rewrite rules. These suffer from the disadvantages described in Section 3.2, but typically perform better than extensionally represented rules, as described in Section 3.10.

Our generic pattern-matching function is a variation on the generic unification functions of Jansson and Jeuring (1998) and Sheard (2001). A generalisation of our library to full unification is possible, but probably hard to keep user-friendly as unification results may contain metavariable occurrences that can then no longer be hidden from the user. Adapting our library to use mutable variables to improve performance, as in Sheard’s work, should be relatively straightforward.

Brown and Sampson (2008) implement generic rewriting using the Scrap Your Boilerplate-library. Patterns are described in a special-purpose datatype that does
not depend on the type of values being rewritten. In contrast to our system, rules are not typed and hence ill-typed rules are only detected at run time.

There exist a number of programming languages built on top of the rewriting paradigm, such as ELAN (Borovansky et al., 2001), OBJ (Goguen and Malcolm, 1997), ASF+SDF (Van Deursen et al., 1996), and Stratego (Bravenboer et al., 2008). Instead of built-in support for rewriting, we focus on how to support rewriting in a mainstream strongly typed functional language by providing a library.

3.12 Conclusion

We have presented a library for datatype-generic term rewriting. Our library overcomes problems in previous generic rewriting libraries: users do not have to adapt or manually extend the datatypes that are used to represent terms; they do not need knowledge of the internals of the library; and they can document, test, and analyse their rewrite rules. The performance of our library is not as good as that of hand-written, datatype-specific rewrite functions, but we think the loss of performance is acceptable for many applications.

In contrast to rewrite rules that are defined using an intensional representation, our library requires that rule synthesisers do not ‘cheat’ by inspecting their metavariable arguments. Concretely, we do not allow arbitrary function applications in the right-hand side of a rule template, but unfortunately this restriction cannot be enforced statically.

There is ongoing work on generating test data for rewrite rules generically. That is, the left-hand side of a rewrite rule can be used as a template for test-data generation to improve testing coverage. We plan to use this approach in a testing framework that is to be shipped with our library.
Dynamic typing and
genralised algebraic datatypes

Abstract

We present a typical synergy between dynamic typing and generalised algebraic datatypes. The former concept allows values to be wrapped together with their type in a uniform package, deferring type checking until run time. Such a dynamic value can be unwrapped by pattern matching and specifying the expected type. The latter concept allows for the explicit specification of constructor types to enforce their structural validity. In contrast to algebraic datatypes, generalised algebraic datatypes are heterogeneous structures since each constructor type is implicitly universally quantified. Unfortunately, pattern matching only enforces structural validity and does not provide information on how the types of the constructors are instantiated. Consequently, functions that manipulate such values are cumbersome due to boilerplate type representation administration. In this chapter we focus on improving such functions by providing a new annotation on generalised algebraic datatype values via a natural synergy with dynamic typing. We motivate the need for the synergy in the context of an update function on \(\lambda\)-terms and formally define a semantics for the new annotation.

4.1 Introduction

Types play an important role in statically typed functional languages such as Clean and Haskell. Static typing is used to prevent erroneous behaviour at run time. Moreover, more efficient code can be generated using the knowledge provided by the types at compile time. However, when for example input is obtained from a user, some types will only be known at run time. Using dynamic typing, monomorphic (Abadi et al., 1991) and polymorphic (Leroy and Mauny, 1993; Abadi et al., 1995; Pil, 1997) values can be wrapped together with their type in a black box. A dynamic value is unwrapped by pattern matching and specifying the expected type in a function definition, instead of specifying the type explicitly in its signature. This approach defers type checking until run time, exactly when the final required type information is made available. Fortunately, this does not take place at the cost of the advantage of static typing since it is guaranteed that when pattern matching succeeds, the unwrapped value can be used in a type-safe fashion from there on. Of course, pattern matching can fail and cause a run-time error, but this is not different from conventional pattern matching.
Algebraic datatypes allow us to inductively define structures. Unfortunately, this does not allow us to enforce structural validity. This restriction is relieved with the arrival of generalised algebraic datatypes (Cheney and Hinze, 2003; Xi et al., 2003; Peyton Jones et al., 2006) by allowing constructors to explicitly dictate their types. On the one hand, this prevents us from constructing ill-structured (i.e., ill-typed) values, and on the other hand this ensures structural validity once such a value is pattern matched. In contrast to algebraic datatypes, generalised algebraic datatypes are heterogeneous structures since each constructor type is implicitly universally quantified. Pattern matching such a value only introduces information regarding the structure of the constructor types, effectively hiding information on how these types are instantiated. However, more information on their instantiation is often required, typically in functions that manipulate such values. Conventional approaches to this problem are cumbersome, due to boilerplate type representation administration.

The main goal of this chapter is to define a type-safe update function on generalised algebraic datatypes by providing a new annotation on these values via a natural synergy with dynamic typing. First, we elaborate on both dynamic typing and generalised algebraic datatypes (Section 4.2). Our contributions are the following:

- We motivate the need for the synergy in the context of an update function on \( \lambda \)-terms (Section 4.3).
- We formally define a semantics for the new annotation (Section 4.4).

We discuss related work (Section 4.5) and conclude with a discussion on future work and other applications of this technique (Section 4.6). In this chapter we use Clean’s dynamic typing and Haskell’s generalised algebraic datatypes. For the sake of presentation, our examples use Haskell syntax, augmented with Clean’s notation for dynamic typing.

## 4.2 Preliminaries

We first discuss dynamic typing (Section 4.2.1) and generalised algebraic datatypes (Section 4.2.2).

### 4.2.1 Dynamic typing

The advantage of statically typed languages is that types are verified at compile time, preventing erroneous behaviour at run time due to ill-typed values. However, static typing sometimes does not suffice since a type might only be known at run time. Using dynamic typing, values are wrapped in a black box, not exposing the type of the contents to the ‘outside’ world. But unlike existential types (Läuffer and Odersky, 1994), both the value and its type can be unwrapped by pattern matching the black box, thereby obtaining a value of the matched content type.

In Clean, the keyword `dynamic` provides the mechanism to wrap a value together with its type in a dynamic value (Vervoort and Plasmeijer, 2003), obtaining a value of type `Dynamic`:
4.2 Preliminaries

\[
\text{wrapInt} :: \text{Int} \to \text{Dynamic}
\]
\[
\text{wrapInt } x = \text{dynamic } x
\]

Unwrapping the wrapped integer value is achieved by pattern matching on the dynamic value using the :: annotation, thereby specifying the expected type:

\[
\text{unwrapInt} :: \text{Dynamic} \to \text{Int}
\]
\[
\text{unwrapInt } (x :: \text{Int}) = x
\]
\[
\text{unwrapInt } (x :: \text{String}) = \text{stringToInt } x
\]
\[
\text{unwrapInt } _- = 0
\]

The first arm of the function pattern matches on a value \(x\) of type \text{Int} in the dynamic value. If this is the case, the value is returned unchanged. However, the value found in the dynamic value is possibly a string and has to be converted to an integer first. Due to run-time type checking, the dynamic pattern match can fail in case the wrapped value is not of the type \text{Int} or \text{String}. It is our responsibility to provide a catch-all arm which either returns a default value or a run-time error message.

Instead of defining a function for each value type that is turned into a dynamic value, we define a single function:

\[
\text{wrap} :: \text{TC } \alpha \Rightarrow \alpha \to \text{Dynamic}
\]
\[
\text{wrap } x = \text{dynamic } x
\]

Since this function is polymorphic in the argument type, we require the context to provide the type code (i.e., the value representation of the type) of \(\alpha\) which is stored together with the value \(x\), using Clean’s built-in \text{TC} class constraint. Besides a description of the type, a type code also contains the definitions of any type involved, because dynamic values can be (de)serialised across modules and verifying name equivalence in a dynamic pattern match does not suffice. Consequently, \text{TC} instances are only available for nonabstract types.

In contrast to Haskell, Clean supports type-dependencies in dynamic typing (Pil, 1999), which allows us to use pattern variables in the type that we specify in a dynamic pattern match:

\[
\text{unwrap} :: \text{TC } \alpha \Rightarrow \text{Dynamic} \to \alpha
\]
\[
\text{unwrap } (x :: \alpha^\wedge) = x
\]
\[
\text{unwrap } _- = \text{error } "\text{unexpected type}"
\]

We require \(x\) to be of type \(\alpha\) and refer to the same variable in the result type of \text{unwrap} using the \(^\wedge\) annotation. This causes both types to be coerced automatically at run time. Therefore, a type code is required for \(\alpha\) such that it can be compared with the type code obtained from the dynamic pattern match. The context in which this function is used determines which type code is to be provided.

Pattern variables can also be used to enforce type equality, for example, to define function application of dynamic values:

\[
\text{apply} :: \text{TC } \beta \Rightarrow \text{Dynamic} \to \text{Dynamic} \to \text{Maybe } \beta
\]
\[
\text{apply } (f :: a \to \beta^\wedge) (x :: a) = \text{Just } (f \ x)
\]
\[
\text{apply } _- _- = \text{Nothing}
\]
The dynamic pattern matches in the first arm share the same scope. Therefore, they only succeed once the argument type of the function matches the type of the argument. Because the result type of the function in the first dynamic pattern match refers to $\beta$ in the result type of \texttt{apply}, a type code is required for this type. As an example, the following expression yields \texttt{Just 1}:

\[
\text{apply (dynamic } \texttt{fst) (dynamic (1, "2"))}
\]

But, the next expression results in \texttt{Nothing} since the argument is not a pair:

\[
\text{apply (dynamic } \texttt{fst) (dynamic 1)}
\]

Finally, dynamic typing preserves the lazy behaviour of functional programs:

\[
\text{apply (dynamic } \texttt{fst) (dynamic (1, ⊥))}
\]

Although the value $\bot$ is part of the tuple that is wrapped in a dynamic value, it is not evaluated when being (un)wrapped and \texttt{Just 1} is returned.

### 4.2.2 Generalised algebraic datatypes

Algebraic datatypes are an oft-used abstraction in functional languages since they provide an inductive approach to defining complex structures by enumerating the alternatives of a type and the associated fields. For example, in Haskell, such a datatype that represents $\lambda$-terms is defined as follows:

\[
\text{data Lam = Undef } | \text{ Const Value } | \text{ App Lam Lam}
\]

The \texttt{Undef} constructor has no fields, while the \texttt{Const} constructor has a single field for a value. The \texttt{App} constructor has two fields, which both can be any term. The values are enumerated by another algebraic datatype:

\[
\text{data Value = VInt Int } | \text{ VFun (Value } \rightarrow \text{ Value)}
\]

Next, we define an evaluation function:

\[
\text{eval :: Lam } \rightarrow \text{ Value}
\]

\[
\text{eval Undef } = \bot
\]

\[
\text{eval (Const } x) = x
\]

\[
\text{eval (App } f x) = \text{ case eval } f \text{ of}
\]

\[
\text{VFun } f \rightarrow f \text{ (eval } x)
\]

\[
\bot \rightarrow \text{ error "expected a function"
}\]

The arms for \texttt{Undef} and \texttt{Const} are straightforward. However, since nothing prevents us from constructing ill-typed terms, the arm for \texttt{App} has to ensure that its first field actually evaluates to a function.

With the arrival of generalised abstract datatypes, we are able to enforce structural validity by providing an explicit type signature to each constructor. Consequently, such a datatype imposes a heterogeneous structure since all constructor types are implicitly universally quantified. We illustrate its use by defining the \texttt{Lam} type again, this time describing typed $\lambda$-terms:
\textbf{data Lam :: \(\star \rightarrow \star\) where}

\begin{tabular}{l}
Undef :: Lam \(\alpha\) \\
Const :: \(\alpha \rightarrow\) Lam \(\alpha\) \\
App :: Lam \((\alpha \rightarrow \beta) \rightarrow\) Lam \(\alpha \rightarrow\) Lam \(\beta\)
\end{tabular}

The \(\text{Lam}\) type is parameterised by the result type of the term once it is evaluated. With each constructor, we explicitly specify its result type. The \text{Undef} constructor represents an undefined value. Since its result type \(\alpha\) is free and not bound by any fields, it can be unified with any other type. The \text{Const} constructor lifts any value to the \(\text{Lam}\) type. The \text{App} constructor is more explicit about the types of its two fields. The argument type of the function term must match the type of the argument term. Then, its result type is the result type of the function term. The explicit constructor types prevent us from constructing ill-typed terms.

Consider the following representation of a term \(\bot 1\):

\[\text{App Undef (Const 1)}\]

Since the return type of the \text{Undef} constructor can be anything, it is instantiated to a function as \text{App} requires, thereby returning a value of type \(\text{Lam}\ \alpha\). When we provide a term that does not return a function, the term becomes ill typed. For instance, this is the case with the following representation of a term \(0 1\):

\[\text{App (Const 0) (Const 1)}\]

A more useful example actually applies a function, for example the term \(\text{abs 1}\):

\[\text{App (Const abs) (Const 1)}\]

This term is well typed and evaluates to a value of type \(\text{Lam}\ \text{Int}\).

Type information described in the type of the constructors is also employed when the constructors are pattern matched in a function definition. Since only well-typed terms can be constructed, we can now safely and concisely define the evaluation function:

\[
\begin{align*}
\text{eval} & :: \text{Lam} \ \alpha \rightarrow \alpha \\
\text{eval Undef} & = \bot \\
\text{eval (Const} \ x \text{)} & = x \\
\text{eval (App} \ f \ x \text{)} & = \text{eval} \ f \ (\text{eval} \ x)
\end{align*}
\]

The result type of the function depends on the term that is evaluated. Each constructor dictates the type of its fields as well as the result type. For example, evaluating the first field of the \text{App} constructors returns a function, which can safely be applied to its evaluated second field. However, be aware that the exact types of these fields are not known since \(\text{Lam}\) is a heterogeneous structure.

\section*{4.3 Motivation}

In this section, we motivate the need for the typical synergy between dynamic typing and generalised algebraic datatypes in the context of an update function on \(\lambda\)-terms (Section 4.3.1). Next, we discuss the problems of the conventional approach (Section 4.3.2) and how the synergy elegantly improves on these issues using a new annotation on generalised algebraic datatype values (Section 4.3.3).
4.3.1 Setting the scene

As the running example, we use the definition from Section 4.2.2 that represents typed λ-terms. Our goal is to define an update function that takes such a term, and updates a field of a constructor at a specified position with a new value. Then, the desired type of the update function becomes:

\[ \text{update} :: \text{Lam} \, \alpha \rightarrow \text{Path} \rightarrow \beta \rightarrow \text{Lam} \, \alpha \]

The argument and result type of the function are the same since we only consider updates that do not affect the top-level type of the term. However, an update can change the structure. The path depicts the location of the update in the heterogeneous structure:

\[ \text{type Path} = [\text{Int}] \]

The path is represented as a list of integers. The length of the list indicates the recursive level (where the empty list is the root) of the target and each value the field (where 0 is the first field) that must be considered. Since the path possibly dictates an update anywhere in the heterogeneous structure, the type of the new value is unrestricted. Hence, the challenge we face lies in only allowing type-safe updates.

4.3.2 Conventional approach

The conventional approach to this problem makes extensive use of explicit type equality proofs (Baars and Swierstra, 2002; Cheney and Hinze, 2002). By comparing the value representations of the type of the old and new value, a proof of type equality can be obtained to ensure only type-safe updates.

First, we modify our original Lam definition from Section 4.2.2 to the following:

\[
\text{data Lam}_R :: \star \rightarrow \star \text{ where} \\
\quad \text{Undef}_R :: \text{Lam}_R \, \alpha \\
\quad \text{Const}_R :: \text{RepOf} \, \alpha \rightarrow \text{Lam}_R \, \alpha \\
\quad \text{App}_R :: \text{RepOf} \, (\text{Lam}_R \, (\alpha \rightarrow \beta)) \rightarrow \text{RepOf} \, (\text{Lam}_R \, \alpha) \rightarrow \text{Lam}_R \, \beta
\]

The difference is that the types of the constructor fields now include a type representation:

\[ \text{type RepOf} \, \alpha = (\alpha, \text{Rep} \, \alpha) \]

The Rep type enumerates the possible types, including the integer type, the function type, and the LamR type:

\[
\text{data Rep} :: \star \rightarrow \star \text{ where} \\
\quad \text{RInt} :: \text{Rep} \, \text{Int} \\
\quad \text{RFun} :: \text{Rep} \, \alpha \rightarrow \text{Rep} \, \beta \rightarrow \text{Rep} \, (\alpha \rightarrow \beta) \\
\quad \text{RLam}_R :: \text{Rep} \, \alpha \rightarrow \text{Rep} \, (\text{Lam}_R \, \alpha)
\]

The Rep type is only a witness of a type, for example, the type LamR (Int → Int) is witnessed by the corresponding value RLamR (RFun RInt RInt). Note that the
representation type only reflects monomorphic types. Given such witnesses, we are able to construct the actual proof that the types of such \textit{Rep} values are the same. Such a proof is constructed by the following generalised algebraic datatype:

\[
data Equal :: \star \rightarrow \star \rightarrow \star 
\text{where} 
Refl :: Equal \alpha \alpha
\]

The \textit{Equal} type consists of a single constructor \textit{Refl}, one that proves that both of the type arguments are the same. Then, we define a type equality function that performs a point-wise comparison of type representations, using Haskell’s \texttt{do}-notation:

\[
eq_R :: \text{Rep} \alpha \rightarrow \text{Rep} \beta \rightarrow \text{Maybe} \ (\text{Equal} \alpha \beta) 
\]

\[
eq_R \text{RInt} \text{RInt} = \text{Just Refl} 
\]

\[
eq_R (\text{RFun} \ x \ 1 \ x \ 2) (\text{RFun} \ y \ 1 \ y \ 2) = \text{do} \begin{cases} 
\text{Refl} \leftarrow \text{eq}_R x_1 \ y_1 \\
\text{Refl} \leftarrow \text{eq}_R x_2 \ y_2 \\
\text{Just Refl} 
\end{cases} 
\]

\[
eq_R (\text{RLam}_R \ x) (\text{RLam}_R \ y) = \text{do} \begin{cases} 
\text{Refl} \leftarrow \text{eq}_R x \ y \\
\text{Just Refl} 
\end{cases} 
\]

\[
eq_R - - = \text{Nothing} 
\]

Given two \textit{Rep} values, this function either returns \text{Just Refl} if the type representations are the same, thereby implicitly indicating that the types \(\alpha\) and \(\beta\) are the same as well, or \text{Nothing}. In the arms for \textit{RFun} and \textit{RLam}_R we explicitly pattern match the result of the recursion to obtain its type equality proof. Finally, we define a catch-all arm which returns \text{Nothing} for \textit{Rep} values that are not equal.

Then, using the modified \textit{Lam} definition and a type representation added to the new value, we are finally able to define our update function:

\[
\text{update}_R :: \text{Lam}_R \alpha \rightarrow \text{Path} \rightarrow \text{RepOf} \beta \rightarrow \text{Lam}_R \alpha 
\]

\[
\text{update}_R \text{Undef}_R - - = \text{Undef}_R 
\]

\[
\text{update}_R (\text{Const}_R \ (x, rx)) \ [0] \ (y, ry) = \begin{cases} 
\text{case eq}_R rx \ ry \ of \\
\text{Just Refl} \rightarrow \text{Const}_R \ (y, ry) \\
\text{Nothing} \rightarrow \text{Const}_R \ (x, rx) 
\end{cases} 
\]

\[
\text{update}_R (\text{App}_R \ (f, rf) \ x) \ [0] \ (y, ry) = \begin{cases} 
\text{case eq}_R rf \ ry \ of \\
\text{Just Refl} \rightarrow \text{App}_R \ (y, ry) \ x \\
\text{Nothing} \rightarrow \text{App}_R \ (f, rf) \ x 
\end{cases} 
\]

\[
\text{update}_R (\text{App}_R f \ (x, rx)) \ [1] \ (y, ry) = \begin{cases} 
\text{case eq}_R rx \ ry \ of \\
\text{Just Refl} \rightarrow \text{App}_R f \ (y, ry) \\
\text{Nothing} \rightarrow \text{App}_R f \ (x, rx) 
\end{cases} 
\]

\[
\text{update}_R (\text{App}_R f \ (x, rx)) \ (0 : p) \ y = \text{App}_R \ (\text{update}_R f \ (p \ y, rf)) \ x 
\]

\[
\text{update}_R (\text{App}_R f \ (x, rx)) \ (1 : p) \ y = \text{App}_R f \ (\text{update}_R x \ (p \ y, rx)) 
\]

\[
\text{update}_R x - - = x 
\]

In the arm for \textit{Undef}_R there is nothing left to do, we simply return the value unchanged. The \textit{Const}_R is the first interesting case, since we have to verify that the types match, by testing the equality of the \textit{Rep} values. Once these values are the same, we try to construct a proof that \(\alpha\) and \(\beta\) are equal types by pattern matching on the \textit{Refl} constructor. Then, in the arms for \textit{App}_R we use the same
approach and either replace its first or second field, or dispatch on the head of
the path and continue to recurse in either of its fields. Finally, a catch-all arm is
included to return the original term once the provided path is incorrect. Every
application of this function explicitly includes type representations:

\[
update_R \left( Const_R \left( \text{abs, RFun RInt RInt} \right) \right) [0] \left( \text{neg, RFun RInt RInt} \right)
\]

This application results in the following:

\[
Const_R \left( \text{neg, RFun RInt RInt} \right)
\]

Although this approach guarantees type-safe updates, it is not a very elegant
definition. First of all, the invasive inclusion of \textit{Rep} values in the datatype clutters
the update function with type equality witnesses and manual proofs. Moreover, the
types of the values that are updated have to be known beforehand since these are
enumerated in the \textit{Rep} type and traversed in the type equality function. Above all,
this approach is not expected to scale to more complex structures and functions.

4.3.3 The synergy

The conventional approach requires us to carry around type representations which
are used to convince the type checker of type equality. When we look back at Sec-
tion 4.2.1, we notice that this is actually what Clean’s \textit{TC} type class provides.
We propose to adapt the original \textit{Lam} definition from Section 4.2.2 again:

\[
data \text{Lam}_T :: \star \rightarrow \star \text{ where } \\
\text{Undef}_T :: \text{Lam}_T \alpha \\
\text{Const}_T :: \text{TC } \alpha \Rightarrow \alpha \rightarrow \text{Lam}_T \alpha \\
\text{App}_T :: \left( \text{TC } \alpha, \text{TC } \beta \right) \Rightarrow \text{Lam}_T \left( \alpha \rightarrow \beta \right) \rightarrow \text{Lam}_T \alpha \rightarrow \text{Lam}_T \beta
\]

Instead of including \textit{Rep} values, we include \textit{TC} class constraints with the con-
structors that can be updated. Then, we define the update function as follows
using the new \( \cdot:G \) annotation on the fields of its constructors:

\[
update_T :: \text{TC } \beta \Rightarrow \text{Lam}_T \alpha \rightarrow \text{Path} \rightarrow \beta \rightarrow \text{Lam}_T \alpha \\
update_T \text{ Undef}_T = \text{ Undef}_T \\
update_T \left( \text{Const}_T \left( x ::G \beta^\land \right) \right) [0] \ y = \text{ Const}_T \ y \\
update_T \left( \text{App}_T \left( f ::G \beta^\land \right) x \right) [0] \ y = \text{ App}_T \ y \ x \\
update_T \left( \text{App}_T \ f \left( x ::G \beta^\land \right) \right) [1] \ y = \text{ App}_T \ f \ y \\
update_T \left( \text{App}_T \ f \ x \right) \ (0 : p) \ y = \text{ App}_T \ \left( update_T \ f \ p \ y \right) \ x \\
update_T \left( \text{App}_T \ f \ x \right) \ (1 : p) \ y = \text{ App}_T \ f \ \left( update_T \ x \ p \ y \right) \\
update_T \ x = \ x
\]

Let us take a look at the differences between this update function and the conven-
tional definition \( update_R \) from Section 4.3.2. First of all, this function operates on
the \textit{Lam}_T type that is decorated with \textit{TC} constraints, and its type contains a \textit{TC}
constraint to obtain a type code for the new value of type \( \beta \). Although the update
function was intended to be polymorphic at first, this constraint only forbids ab-
abstract types to occur as new values, as discussed earlier in Section 4.2.1. Another
difference is that the function is no longer cluttered with verbose type equality
witnesses and manual proofs. Instead, the fields of the constructors are annotated
using the new ::\# annotation, thereby accessing the instantiation information. For
example, in the arm for \( Const_T \), the annotation denotes that \( x \) is of type \( \beta \), or
even more specific, the type of the new value as determined by the context in
which this function is used. Note that the catch-all arm of the function now also
takes care of any failing tests for type equality. Comparing the use of this update
function to the conventional approach emphasises the elegance of our approach:

\[
update_T \ (Const_T \ abs) \ [0] \ neg
\]

This application of the new update function yields the following:

\[
Const_T \ neg
\]

Instead of explicitly providing type representations and equality proofs, it is now
the context that implicitly determines which fields are eligible for an update.

4.4 Semantics

In this section we present a formal semantics for the new annotation. We define a
core functional language and an extension for the annotation (Section 4.4.1). Then,
we describe the translation from the extended language to the core language by
means of an example (Section 4.4.2), followed by a formal definition (Section 4.4.3).

4.4.1 Formal language

The functional core language FC, which forms the basis of our semantics, is depicted
in Figure 4.1. It is a common subset of Clean and Haskell, extended with dynamic
typing and generalised algebraic datatypes. An FC program consists of zero or
more datatype declarations and function declarations. A datatype is either a type
synonym, an algebraic datatype or a generalised algebraic datatype. A type comes
in three flavours: a qualified type, a base type, and an annotation type. A qualified
type only includes the TC constraint, to facilitate dynamic typing, where we write
\( \tau \) as a shorthand for the qualified type \( \cdot \Rightarrow \tau \) with no constraints. Second, a base
type comprises the polymorphic types. Very much like for a base type, we define a
separate annotation type, but one that also allows the use of the \( ^\wedge \) annotation. A
named function is defined by its type and body. Amongst the well-known expres-
sions, our language supports the case construct to pattern match (dynamic) values,
typically the arguments of a function. In the language of patterns we distinguish a
nested pattern from a base pattern to prepare for the language extension. Finally,
we do not explicitly include lists and tuples of arbitrary arity in the language of
expressions, patterns and types, since these are easily realised through predefined
algebraic datatypes. We do not provide operational semantics and typing for the
core language since these have been studied in-depth elsewhere (Cartwright and
Donahue, 1982; Abadi et al., 1995; Cheney and Hinze, 2003).

Next, we define the extended language FC+ that allows us to use the new an-
notation on generalised algebraic datatype values, as shown in Figure 4.2. For the
4 Dynamic typing and generalised algebraic datatypes

\[
\begin{align*}
\text{(program)} & \quad \pi := \delta \phi \\
\text{(datatype declaration)} & \quad \delta := \text{type } T \alpha = \tau \\
& \quad \text{| data } T \alpha = C \tau \\
& \quad \text{| data } T :: \kappa \text{ where } C :: \sigma \\
\text{(qualified type)} & \quad \sigma := TC \alpha \Rightarrow \tau \\
\text{(base type)} & \quad \tau := \alpha \mid \text{Int} \mid T \\
& \quad \text{| } \tau_1 \tau_2 \mid \tau_1 \rightarrow \tau_2 \\
& \quad \text{| Dynamic} \\
\text{(annotation type)} & \quad \omega := \alpha \mid \alpha \land \mid \text{Int} \mid T \\
& \quad \text{| } \omega_1 \omega_2 \mid \omega_1 \rightarrow \omega_2 \\
& \quad \text{| Dynamic} \\
\text{(kind)} & \quad \kappa := \star \mid \kappa_1 \rightarrow \kappa_2 \\
\text{(function declaration)} & \quad \phi := \text{fix } f :: \sigma = \epsilon \\
\text{(expression)} & \quad \epsilon := \bot \mid i \mid x \mid C \\
& \quad \text{| } \epsilon_1 \epsilon_2 \mid \lambda x \rightarrow \epsilon \mid \text{case } \epsilon_s \text{ of } \rho \rightarrow \tau \\
& \quad \text{| dynamic } e :: \omega \\
\text{(nested pattern)} & \quad \rho := \varrho \mid C \overline{p} \\
\text{(base pattern)} & \quad \varrho := \_ \mid i \mid x \\
& \quad \text{| } x :: \omega
\end{align*}
\]

Figure 4.1: The core language \( \mathcal{FC} \)

sake of simplicity, we only allow the new annotation to occur on the top level of a constructor field pattern. However, nested patterns can be easily achieved by nesting case expressions. We redefine patterns in \( \mathcal{FC} \) case expressions to be either a base pattern or a constructor with field patterns. Then, a pattern in a constructor field is either an original nested pattern, or an identifier annotated with a type.

As an example, we define \( \text{update}_T \) from Section 4.3.3 in the \( \mathcal{FC}+ \) language:

\[
\begin{align*}
\text{fix } \text{update}_T :: TC \beta & \Rightarrow \text{Lam}_T \alpha \rightarrow \text{Path} \rightarrow \beta \rightarrow \text{Lam}_T \alpha = \\
& \lambda x \rightarrow \lambda p \rightarrow \lambda y \rightarrow \text{case } (x, p) \text{ of } \\
& \quad (\text{Undef}_T, \_) \rightarrow \text{Undef}_T \\
& \quad (x, (0 : [])) \rightarrow \text{case } x \text{ of } \\
& \quad \quad \text{Const}_T (x :: \varrho \beta^\land) \rightarrow \text{Const}_T y \\
& \quad \quad \text{App}_T (f :: \varrho \beta^\land) x \rightarrow \text{App}_T y x \\
& \quad \quad \quad \rightarrow x \\
& \quad (x, (1 : [])) \rightarrow \text{case } x \text{ of } \\
& \quad \quad \quad \text{App}_T f (x :: \varrho \beta^\land) \rightarrow \text{App}_T f y \\
& \quad \quad \quad \quad \rightarrow x \\
& \quad (\text{App}_T f x, (0 : p)) \rightarrow \text{App}_T (\text{update}_T f p y) x \\
& \quad (\text{App}_T f x, (1 : p)) \rightarrow \text{App}_T (\text{update}_T x p y) \\
& \quad \quad \rightarrow x
\end{align*}
\]
(expression) \[ \epsilon ::= \cdots \]
| \cdots | \textbf{case} \[ \epsilon_s \textbf{ of } \overrightarrow{\theta} \rightarrow \epsilon \]
| \cdots |

(pattern) \[ \theta ::= \varrho | C \overrightarrow{\vartheta} \]

(field pattern) \[ \vartheta ::= \rho | x ::^{G} \omega \]

Figure 4.2: The extended language \( \mathcal{FC}^+ \)

While being slightly more verbose than the original definition, a translation from a Clean or Haskell definition is easily made. Note that the definitions of the \emph{Path} and \emph{Lam}\_T type from Section 4.3.1 and Section 4.3.3 respectively do not change in the formal model.

### 4.4.2 Intuition

The general idea behind the translation is to take each generalised algebraic datatype and translate it to an extended parallel definition in which only constructor fields that are annotated in the program, are decorated with additional type information. A conversion function takes care of inserting type information in the original definition and the \( ::^{G} \) annotations are translated such that it accesses this information.

For example, the \( \text{Lam}\_T \) type from Section 4.3.3 translates to the following:

```haskell
data Lam\_T\^:: \star \rightarrow \star \text{ where }
  Undef\_T\^:: Lam\_T\^\alpha
  Const\_T\^:: TC \alpha \Rightarrow TypeOf \alpha \rightarrow Lam\_T\^\alpha
  App\_T\^:: (TC \alpha, TC \beta) \Rightarrow
      TypeOf (Lam\_T (\alpha \rightarrow \beta)) \rightarrow TypeOf (Lam\_T \alpha) \rightarrow Lam\_T\^\beta
```

The extended definition, as well as its constructors, is given a new name. Since all fields of the constructors are annotated in the update function from Section 4.3.3, all fields of the \( \text{Const}\_T\^ \) and \( \text{App}\_T\^ \) constructor now contain a typed value. Note that in order to only have to translate patterns instead of complete functions later on, the addition of type information is nonrecursive:

```haskell
type TypeOf \alpha = (\alpha, \text{Dynamic})
```

A typed value is simply the original value paired with its type stored in a dynamic value. As a \emph{Dynamic} can contain a value of any type, not necessarily the type \( \alpha \), we use the following function to obtain correctness by construction:

```haskell
fix typeOf :: TC \alpha \Rightarrow \alpha \rightarrow TypeOf \alpha =
\lambda x \rightarrow (x, \text{dynamic } \bot :: \alpha^\wedge)
```

Since we only need the type of a value, it suffices to wrap \( \bot \) instead of an actual value. As described in Section 4.2.1, the \( ^\wedge \) annotation refers to context-dependent
type information. Meaning, the context in which typeOf is used determines the type that is stored in the dynamic value. Note that constructors can contain generalised algebraic datatype values, like \( App^\circ_T \), which consequently requires such types to be stored in a dynamic value. Unfortunately, type code facilities are yet to be defined for generalised algebraic datatypes. Although they greatly complicate the type system (Peyton Jones et al., 2006; Schrijvers et al., 2009), we hypothesise that wrapping such values in dynamic values is straightforward since unification of their type codes is not different from ordinary algebraic datatypes.

Then, the conversion from the original to the extended definition injects the type information in constant time using the function typeOf:

\[
\text{fix } toLam^\circ_T :: Lam_T \alpha \to Lam^\circ_T \alpha = \\
\lambda x \to \text{case } x \text{ of} \\
\quad \text{Undef}_T \to \text{Undef}^\circ_T \\
\quad \text{Const}_T x \to \text{Const}^\circ_T (\text{typeOf } x) \\
\quad \text{App}_T f x \to \text{App}^\circ_T (\text{typeOf } f) (\text{typeOf } x)
\]

The conversion only renames the \( \text{Undef}_T \) constructor since it has no fields. The fields of the \( \text{Const}_T \) and \( \text{App}_T \) constructor are extended with their types. As the function typeOf dictates, this requires a type code for the field types. The translation relies critically on this assumption, which is enforced by only considering a \( \text{fc}^+ \) program well formed, if and only if, each constructor has \( \text{TC} \) constraints on every type variable occurring in its annotated fields. Fortunately, as mentioned before in Section 4.2.1, the \( \text{TC} \) constraint is easily discharged for any nonabstract type, which only forbids the use of the new annotation in combination with abstract types.

Finally, we define the translation of the actual \( ::^G \) annotation, accessing the inserted type information. For example, the \( \text{fc}^+ \) function update\(_T\), as defined in Section 4.4.1, is translated to \( \text{fc} \):

\[
\text{fix } update^\circ_T :: \text{TC } \beta \Rightarrow Lam_T \alpha \to \text{Path} \to \beta \Rightarrow Lam_T \alpha = \\
\lambda x \to \lambda p \to \lambda y \to \text{case } (x, p) \text{ of} \\
\quad \ldots \\
\quad (x, (0 : [])) \to \text{case } toLam^\circ_T x \text{ of} \\
\qquad \text{Const}^\circ_T (x, _:: \beta^\wedge) \to \text{Const}_T y \\
\qquad \text{App}^\circ_T (f, _:: \beta^\wedge) (x, _) \to \text{App}_T y x \\
\qquad \text{ } \rightarrow x \\
\qquad \ldots \\
\qquad (x, (1 : [])) \to \text{case } toLam^\circ_T x \text{ of} \\
\qquad \text{App}^\circ_T (f, _) (x, _:: \beta^\wedge) \to \text{App}_T f y \\
\qquad \text{ } \rightarrow x \\
\quad \ldots
\]

The conversion from the original to the extended generalised algebraic datatype is applied to the scrutinee of the case expression. This provides type information in the pattern match, allowing it to interact naturally like a conventional dynamic value, in this case with the type of the function using the \( ^\wedge \) annotation. Note that since the conversion function is specific to a program, and not to each case expression, the fields that do not use the new annotation must discard the inserted type information, such as in the case of \( \text{App}^\circ_T \).
4.4 Semantics

\[
\begin{align*}
\llbracket \pi_{FC^+} \rrbracket & \equiv \pi_{FC} \\
\llbracket \delta \rrbracket & \equiv \delta' \phi^c \\
\llbracket \phi \rrbracket & \equiv \phi' \\
\llbracket \delta \phi \rrbracket & \equiv \delta' \phi^c \phi'^c
\end{align*}
\]  

(T-PROG)

Figure 4.3: Translation of programs

\[
\begin{align*}
\llbracket \delta_{FC^+} \rrbracket & \equiv \delta_{FC} \phi_{FC} \\
\llbracket \text{type } T \alpha = \tau \rrbracket & \equiv \text{type } T \alpha = \tau; \\
\llbracket \text{data } T \alpha = C \tau \rrbracket & \equiv \text{data } T \alpha = C \tau; \
\end{align*}
\]  

(T-DATA-TSYN) (T-DATA-ADT)

\[
\begin{align*}
\neg \text{annotated}(T) \\
\llbracket \text{data } T :: \kappa \text{ where } C :: \sigma \rrbracket & \equiv \text{data } T :: \kappa \text{ where } C :: \sigma; \\
\end{align*}
\]  

(T-DATA-GADT-1)

\[
\begin{align*}
\text{annotated}(T) \\
\llbracket \sigma \rrbracket_C & \equiv \sigma^c \tau; \epsilon^c \\
\delta^c & \equiv \text{data } T^c :: \kappa \text{ where } C^c :: \sigma^c \\
\phi^c & \equiv \text{fix to } T^c :: T \alpha \rightarrow T^c \alpha = \\
\text{parity}(T) & \equiv \alpha \\
\llbracket \text{data } T :: \kappa \text{ where } C :: \sigma \rrbracket & \equiv \text{data } T :: \kappa \text{ where } C :: \sigma \delta^c; \phi^c \\
\end{align*}
\]  

(T-DATA-GADT-2)

Figure 4.4: Translation of datatypes

4.4.3 Formal translation

We continue by defining the formal translation from the extended language FC+ to the core language FC.

Let us begin by translating programs, as depicted by Rule t-prog in Figure 4.3. A program in the FC+ language is translated to the FC language by translating both the datatype declarations and the function declarations.

In Figure 4.4 we give the translation of datatype declarations. Type synonyms and algebraic datatypes are left unchanged, as defined by Rules t-data-tsyn and t-data-adt. We distinguish generalised algebraic datatypes using the metafunction annotated(T) to test if it is pattern matched somewhere in the program using the new annotation (e.g., annotated(LamT) \equiv True). If not, the original definition is returned without any modifications, as defined by Rule t-data-gadt-1. However, an annotated generalised algebraic datatype requires some effort. In Rule t-data-gadt-2, the translation results in the original definition, an extended definition \( \delta^c \) and a conversion function \( \phi^c \). By translating the types of the con-
4 Dynamic typing and generalised algebraic datatypes

\[
\llbracket \sigma_{\text{FC}} \rrbracket C \equiv \sigma_{\text{FC}}; \bar{x}; \bar{\epsilon}_{\text{FC}}
\]

\[
\llbracket \tau \rrbracket C; 0 \equiv \tau^0; \bar{x}; \bar{\epsilon}^0 \quad \text{(T-QTYPE)}
\]

\[
\llbracket TC \alpha \Rightarrow \tau \rrbracket \equiv \llbracket TC \rrbracket \alpha \Rightarrow \tau^0; \bar{x}; \bar{\epsilon}^0
\]

Figure 4.5: Translation of qualified types

\[
\llbracket \tau_{\text{FC}}^+ \rrbracket C; n \equiv \tau_{\text{FC}}; \bar{x}; \bar{\epsilon}_{\text{FC}} \quad \text{(T-TYPE-DATA)}
\]

\[
\llbracket T \rrbracket C; n \equiv T^0; \bar{x}; \bar{\epsilon} \quad \text{(T-TYPE-APP)}
\]

\[
\neg \text{annotated}(C, n) \quad \llbracket \tau_2 \rrbracket C; n+1 \equiv \tau^0_2; \bar{x}_2; \bar{\epsilon}^0_2 \quad \text{(T-TYPE-FUN-1)}
\]

\[
\text{annotated}(C, n) \quad \llbracket \tau_2 \rrbracket C; n+1 \equiv \tau^0_2; \bar{x}_2; \bar{\epsilon}^0_2 \quad \text{(T-TYPE-FUN-2)}
\]

Figure 4.6: Translation of base types

structors, parameterised by the respective constructor name, we obtain extended types together with corresponding pattern variables and expressions that extend these variables. The former is used to define the constructor types of the extended definition, the latter two to define the corresponding conversion function. The metafunction \( \text{tarity}(T) \) provides zero or more fresh type variables, determined by the arity of the type \( T \) (e.g., \( \text{tarity}(\text{Lam}_T) \equiv \alpha \)).

The translation of qualified types, parameterised by a constructor name, is shown in Figure 4.5. A qualified type propagates translation to its base type and includes a parameter for the index of the constructor field type under translation.

In Figure 4.6 we define the parameterised translation of such types, resulting in an extended type, pattern variables and expressions that extends these variables. Since we are only interested in the fields of a constructor type, and the type of an empty constructor is either a type constructor or a type application, Rules \( \text{T-TYPE-DATA} \) and \( \text{T-TYPE-APP} \) result in an unchanged type and no pattern variables or expressions. The function type is the interesting case. If a constructor field is not annotated in the program, as shown in Rule \( \text{T-TYPE-FUN-1} \), it is returned unchanged together with a fresh pattern variable \( x_1 \) and an equal expression. Otherwise, the translation in Rule \( \text{T-TYPE-FUN-2} \) extends the type of the constructor field with additional type information and ensures that the fresh pattern variable is extended as well. In both cases we recurse in the translation by incrementing the second parameter to denote the next constructor field.

Figure 4.7 defines the translation of functions. Rule \( \text{T-FUN} \) shows that only the body is translated; this localises the conversion, leaving its type unchanged.
\[
\begin{align*}
\phi_{FC^+} & \equiv \phi_{FC} \\
\epsilon & \equiv \epsilon' \\
\text{fix } f :: \sigma = \epsilon & \equiv \text{fix } f :: \sigma = \epsilon' \tag{T-FUN}
\end{align*}
\]

Figure 4.7: Translation of functions

\[
\begin{align*}
\epsilon_{FC^+} & \equiv \epsilon_{FC} \\
\bot & \equiv \bot \tag{T-EXP-BOT} \\
i & \equiv i \tag{T-EXP-INT} \\
\epsilon & \equiv \epsilon' \tag{T-EXP-ABS} \\
x & \equiv x \tag{T-EXP-ID} \\
\epsilon_1 \epsilon_2 & \equiv \epsilon'_1 \epsilon'_2 \tag{T-EXP-APP} \\
\text{case } \epsilon_s \text{ of } \theta \to \epsilon & \equiv \text{case } \epsilon'_s \text{ of } \rho \to \epsilon' \tag{T-EXP-CASE-1} \\
x :^\mathcal{G} \alpha \notin \overline{\theta} & \\
\epsilon & \equiv \epsilon' \tag{T-EXP-DYN}
\end{align*}
\]

Figure 4.8: Translation of expressions

The translation of expressions is shown in Figure 4.8. The basic building blocks of expressions: bottom, integers, identifiers, and constructors, are left unchanged, as can be seen in Rules T-EXP-BOT, T-EXP-INT, T-EXP-ID, and T-EXP-CON respectively. Translation of an application is defined by Rule T-EXP-APP and translates both its expressions and Rule T-EXP-ABS defines the translation of an abstraction by translating the body expression. For case expressions, we define two separate rules, testing if one of its patterns uses the new annotation. If not, it suffices to only translate the scrutinee and the expression of each pattern, as defined by Rule T-EXP-CASE-1. Otherwise, Rule T-EXP-CASE-2 defines that the conversion function must be applied to the translated scrutinee. The name of this function is determined by the metafunction \text{btype}(\epsilon_s) which determines the base name of the type of the scrutinee \epsilon_s (e.g., \text{btype}(Const_T \ T_1) \equiv \text{Lam}_T). Furthermore, each
pattern is translated so that the actual use of the annotation is translated. As we will see in a moment, the translation of patterns takes care of renaming the constructors, which is required since the scrutinee is converted to the extended type. Finally, Rule \texttt{t-exp-dyn} defines the translation of a dynamic value, simply translating its expression.

Patterns possibly provide access to the inserted type information, their translation is shown in Figure 4.9. A base pattern is left untouched, as depicted in Rule \texttt{t-pat-base}. In Rule \texttt{t-pat-con}, the constructor in a constructor pattern is renamed and its fields are all translated, parameterised by the name of the original constructor and a metavalue \texttt{index} that provides the index of each constructor field.

In Figure 4.10 we conclude the translation from \texttt{FC} to \texttt{FC} by defining the translation of field patterns, being the language extension itself. Since the conversion function that inserts type information is specific to a program, we have to verify if the current field pattern is ever annotated in the program. Rule \texttt{t-fpat-pat-1} defines that if a field is never annotated, it need not to be translated. Otherwise, the additional information is discarded, as defined by Rule \texttt{t-fpat-pat-2}. The core of the translation is captured by Rule \texttt{t-fpat-ann}. The new annotation is erased by translating it to a dynamic type annotation, yielding a pair that matches the original value and the type stored in the dynamic value.

\section{Related work}

The foundations of structured programming on generalised algebraic datatypes described by Johann and Ghani (2008) provide an elegant approach to defining corresponding algebras. While such algebras provide an abstraction mechanism to define an update function, explicit type representations and type equality proofs
are still required. In Section 4.3.2, we discussed the disadvantages of such an approach. In our work, these representations and proofs are implicitly provided using dynamic typing, which significantly improves the elegance of the function definitions.

Another approach to heterogeneous structures reflects the structure of a value directly in its type (Kiselyov et al., 2004). For example, the type of a heterogeneous list is basically a structure of nested tuples. Then, functions are defined on such structures using the type class mechanism, dispatching on the type structure. To enforce type-safe updates, yet another type class is defined to reflect type equality. Consequently, this approach results in rather verbose definitions since all action takes place on the level of type classes. Since the structure of the types are available, direct manipulation enables type-changing functions. Looking at the type of the update function in Section 4.3.3, our approach seems to forbid any type-changing updates. However, subterms can be replaced by arbitrary complex terms, thereby changing the underlying type structure.

4.6 Conclusion

We have presented a typical synergy between dynamic typing and generalised algebraic datatypes to elegantly define functions that manipulate such values, requiring information on the instantiation of constructor types. Our approach comprises a new annotation on generalised algebraic datatype values and improves upon boilerplate type representation administration in conventional approaches: functions are not cluttered any more with type equality witnesses and manual proofs. Also, by using dynamic typing, we no longer need to maintain a closed enumeration of the used types. Above all, our approach scales up to more complex structures and functions due to its simplicity. We have shown that the language extension is straightforwardly translated to a functional core that supports both dynamic typing and generalised algebraic datatypes.

One of the major limitations in our approach is that the use of type codes limits the use of the new annotation to nonabstract types. It remains future work to define type codes for such types, as well as investigating if dynamic typing can be implemented without requiring type codes as class constraints. This would improve our approach considerably since it will no longer require us to decorate generalised algebraic datatypes beforehand with type code constraints. Also, we plan to verify our hypothesis that storing generalised algebraic datatypes in dynamic values is no different from conventional algebraic datatypes.

Despite these limitations, the translation to dynamic typing provides other opportunities as well, such as type dispatching and enforcing type equality invariants on generalised algebraic datatype values.
Part III

Accommodating dynamic typing
Ad-hoc polymorphism and dynamic typing

Abstract

Static typing in functional languages such as Clean, Haskell, and ML is highly beneficial: it prevents erroneous behaviour at run time and provides opportunities for optimisations. However, dynamic typing is just as important as sometimes types are not known until run time. Examples are exchanging values between applications by deserialisation from disk, input obtained from a user, or obtaining values via a network connection. Ideally, a static type system works in close harmony with an orthogonal dynamic type system: not discriminating between statically and dynamically typed values. In contrast to Haskell’s minimal support for dynamic typing, Clean has an extensive dynamic typing; it adopted ML’s support for monomorphism and polymorphism and added the notion of type dependencies. Unfortunately, ad-hoc polymorphism has been left out over the years. While both ad-hoc polymorphism and dynamic typing have been studied in-depth earlier, their interaction in a statically typed functional language has not been studied before. In this chapter we explore the design space of their interactions.

5.1 Introduction

Static typing is the cornerstone of functional languages such as Clean, Haskell, and ML. It prevents erroneous behaviour at run time by verifying type safety at compile time. Also, it provides opportunities for optimisations by exploiting either user-specified or inferred type information statically.

However, sometimes the type of a value is not known until run time. Typically this is the case when interacting with the ‘outside’ world: exchanging values between applications by deserialisation from disk, input obtained from a user, or obtaining values via a network connection. In such cases, dynamic typing is required to defer type checking until run time. Values are wrapped in a uniform black box, as their type is statically not known, and unwrapped by pattern matching and specifying the expected type. Although type checking can fail at run time when a dynamic value presents an unexpected type, the static type system guarantees that when pattern matching succeeds, the unwrapped value can be used in a type-safe fashion from there on.

While many dispute over choosing either static or dynamic typing, we agree that the solution lies in the middle (Meijer and Drayton, 2004): “Static typing where possible, dynamic typing when needed”. We believe that a statically typed
language is the starting point, extended with an escape to type values dynamically. Ideally, the dynamic type system is orthogonal to the static type system, imposing no restrictions on the value or types that can live in the dynamic world.

Haskell has minimal support for dynamic typing: it only supports monomorphism (Baars and Swierstra, 2002; Cheney and Hinze, 2002). Clean, on the other hand, has a rich and mature dynamic type system; it adopted ML’s support for monomorphism (Abadi et al., 1991) and polymorphism (Leroy and Mauny, 1993; Abadi et al., 1995; Pil, 1997). Additionally, it includes the notion of type dependencies (Pil, 1999). Even generic functions can be applied to dynamic values (Achten et al., 2003; Wichers Schreur and Plasmeijer, 2005). Though, the quest for an orthogonal dynamic type system cannot be completed without proper support for another important concept: ad-hoc polymorphism.

Ad-hoc polymorphism provides an abstraction mechanism to parameterise values with behaviour. The usual suspects are functions for the equality and ordering of values. Whereas in ML ad-hoc polymorphism is modelled via the module system (Wehr and Chakravarty, 2008), Haskell and Clean model ad-hoc polymorphism via type classes which is resolved to a dictionary-passing style at compile time (Wadler and Blott, 1989; Peterson and Jones, 1993). Static type information is crucial in this approach; it is the driving force behind the translation.

Although both ad-hoc polymorphism and dynamic typing have been studied in-depth before, their interaction in a statically typed functional language has not been explored yet. We identify two sides to their interaction. On the one hand, it involves dynamic typing in the world of ad-hoc polymorphism. For instance, when applying a sorting function to a dynamically typed list of values obtained by deserialisation from disk or from a user as input. Obviously, this poses a challenge since ad-hoc polymorphism is resolved at compile time while the type of dynamic values is only known at run time. Typically, this is solved by enumerating all expected types by hand. This resolves ad-hoc polymorphism statically but is cumbersome, prone to errors, and does not scale for evident reasons. On the other hand, the interaction concerns ad-hoc polymorphism in the world of dynamic typing. For example, a sorting function that is deserialised from disk and applied to some statically typed value. Here, the challenge is to extend the existing dynamic typing mechanisms to support ad-hoc polymorphic values.

In this chapter, we explore the design space of the interactions and provide a thorough intuition of the issues involved. While there is a plethora of type class extensions (Peyton Jones et al., 1997), we first only consider type classes in the style of Haskell 98 (Peyton Jones, 2003). We set the scene by giving an overview of conventional ad-hoc polymorphism via type classes in Clean and Haskell (Section 5.2), and dynamic typing in Clean (Section 5.3). Our contributions are the following:

- We describe two complementary approaches to dynamic typing in ad-hoc polymorphism (Section 5.4): container datatypes and dynamic dictionary composition.

- We describe two different approaches to ad-hoc polymorphism in dynamic typing (Section 5.5): dictionary-passing types and type code extension.
• We discuss how several type class extensions affect both sides of the interaction (Section 5.6).

Finally, we elaborate on related work (Section 5.7) and conclude with a brief discussion and future work (Section 5.8).

The examples given in this chapter are defined using Clean syntax. While types in Clean have an explicit arity, we curry the types for the sake of presentation. Also, we explicitly quantify type variables since in some of the examples there are multiple binding sites. An overview of syntactic and semantic differences between Clean and Haskell is given in Chapter 2 and by Achten (2007).

5.2 Ad-hoc polymorphism

We give a brief overview of type classes (Section 5.2.1) and their translation to dictionary-passing style (Section 5.2.2).

5.2.1 Type classes

Consider the following type class for the equality of values:

\[
\text{class} \ Eq \ \alpha \ \text{where} \\
\text{eq} :: \alpha \rightarrow \alpha \rightarrow \text{Bool}
\]

The type class \( Eq \) has a single member, the equality function. We ignore default members for simplicity reasons; these are irrelevant to our approach and only clutter the examples. We provide several instances for this type class:

\[
\text{instance} \ Eq \ \text{Int} \ \text{where} \\
\text{eq} \ x \ y = \text{eqInt} \ x \ y
\]

\[
\text{instance} \ Eq \ [\alpha] \ | \ Eq \ \alpha \ \text{where} \\
\text{eq} \ x \ y = \text{eq} \ (\text{length} \ x) \ (\text{length} \ y) \land \text{and} \ (\text{zipWith} \ \text{eq} \ x \ y)
\]

\[
\text{instance} \ Eq \ (\alpha, \beta) \ | \ Eq \ \alpha \ & \ Eq \ \beta \ \text{where} \\
\text{eq} \ x \ y = \text{eq} \ (\text{fst} \ x) \ (\text{fst} \ y) \land \text{eq} \ (\text{snd} \ x) \ (\text{snd} \ y)
\]

We assume that in the instance for integers, a core equality function \( \text{eqInt} \) is available. In the instance for lists, we require there to be an instance of \( Eq \) for the element type as well since we pairwise compare the elements. Similarly in the instance for pairs, we require instances of \( Eq \) for both element types.

Next, we define a type class for the ordering of values:

\[
\text{class} \ Ord \ \alpha \ | \ Eq \ \alpha \ \text{where} \\
\text{lt} :: \alpha \rightarrow \alpha \rightarrow \text{Bool}
\]

This type class also has a single member function, one that tests if its first argument is ‘less than’ its second argument. We see that the \( Eq \) type class is a superclass of the \( Ord \) class, denoting that for every \( Ord \) instance there must be a corresponding \( Eq \) instance. Again, we define instances for integers, lists, and pairs:
instance Ord Int where
   lt x y = ltInt x y

instance Ord [α] | Ord α where
   lt x y = lt (length x) (length y) \lor or (zipWith lt x y)

instance Ord (α, β) | Ord α & Ord β where
   lt x y = lt (fst x) (fst y) \lor lt (snd x) (snd y)

We assume the presence of a core function \texttt{ltInt} for the ordering of integers. Similar to the instance for \texttt{Eq}, the instances for lists and pairs require instances for their element types. Admittedly, not all of these instances are useful in practice. Here, they merely serve the purpose of illustrating the dictionary-passing style translation.

A typical use of the ordering type class is a sorting function:

\[
\text{sort} :: \forall \alpha . [\alpha] \rightarrow [\alpha] | \text{Ord} \ \alpha
\]

For the sake of brevity we leave its definition abstract. The type of \texttt{sort} reflects that it sorts a list of values, constrained by a context \texttt{Ord} of the element type. Note that since \texttt{Eq} is a superclass of \texttt{Ord}, this makes both the \texttt{lt} function and the \texttt{eq} function available to the sorting function.

### 5.2.2 Dictionary-passing style

Type classes can be translated at compile time to a dictionary-passing style. Each type class definition translates to a dictionary type that captures its members and superclasses. Then, each instance is an instantiation of that dictionary type. For example, in Figure 5.1 we see the dictionary-passing style translation of the \texttt{Eq} and \texttt{Ord} type class and their instances. The \texttt{Eq} type class translates to a record type \texttt{DictEq} that has a single field for its member function \texttt{eq}. The \texttt{DictOrd} record has a field for its member \texttt{lt} and an additional field for the dictionary of its super class \texttt{Eq}. Each of the instance bodies is visible in the instantiations of the dictionary types. The instances that require instances for their element types, such as for lists and pairs, are passed additional dictionaries. In \texttt{dictEqList} and \texttt{dictOrdList} we also see how a concrete dictionary is used to compare the length of the argument lists.

Then, ad-hoc polymorphic values are translated such that they receive additional dictionary arguments. For example, the dictionary-passing type of the sorting function becomes:

\[
\text{sort} :: \forall \alpha . \text{DictOrd} \ \alpha \rightarrow [\alpha] \rightarrow [\alpha]
\]

At every occurrence of the sorting function an additional appropriate dictionary argument is passed implicitly. For instance, consider the following expression:

\[
\text{let } x = [1 \ldots 10] \text{ in eq (sort x) x}
\]

Resolving the occurrences of \texttt{eq} and \texttt{sort} results in the following expression in dictionary-passing style:
5.3 Dynamic typing

Figure 5.1: Dictionary-passing style translation of \textit{Eq} and \textit{Ord}

\begin{verbatim}
let x = [1..10]
in (dictEqList dictEqInt).eq (sort (dictOrdList dictOrdInt) x) x
\end{verbatim}

Here we see that the equality function is translated such that it accesses the appropriate field in the equality dictionary for a list of integers. The sorting function is provided an additional argument, namely, the ordering dictionary for a list of integers.

5.3 Dynamic typing

Next, we give a crash course (not to be taken literally) in Clean’s dynamic type system, discussing monomorphism (Section 5.3.1), polymorphism (Section 5.3.2), type dependencies (Section 5.3.3), and type codes (Section 5.3.4).

5.3.1 Monomorphism

In Clean, dynamic typing allows monomorphic values to be wrapped together with their type in a uniform package of type \textit{Dynamic}, called a dynamic value, using the keyword \texttt{dynamic}:
wrapInt :: Int → Dynamic
wrapInt x = dynamic x :: Int

Using the :: annotation, we explicate the type of the value that is wrapped. The annotation is optional and only required when the type cannot be inferred.

A dynamic value is unwrapped by pattern matching and specifying the expected type using the :: annotation; for example to retrieve an integer value:

unwrapInt :: Dynamic → Int
unwrapInt (x :: Int) = x
unwrapInt (x :: String) = stringToInt x
unwrapInt _ = ⊥

The first arm pattern matches on integer values, returning the value itself if that is the case. If the value found in the dynamic value is a string, we convert it to an integer. As type checking takes place at run time and pattern matching can fail, a catch-all arm is required for totality; either returning a default value or a run-time error message. For the sake of convenience, we choose to return ⊥ for failed dynamic pattern matches in this chapter.

Instead of enumerating every possible type, pattern variables can be used in the type of a dynamic pattern match. Typically, this is used to enforce type equality between dynamic values, for instance in the infamous example of dynamic function application:

dynApp :: Dynamic → Dynamic → Dynamic
dynApp (f :: a → b) (x :: a) = dynamic (f x)
dynApp _ _ = ⊥

Pattern variables, denoted here by roman instead of greek characters, in a single arm definition share the same scope. Therefore, the first arm only succeeds once the argument type of the function matches the type of the argument. Then, the result is wrapped in a dynamic value again.

### 5.3.2 Polymorphism

Besides monomorphic values, polymorphic values can be (un)wrapped as well without any additional effort. For example, a function that does not change the type of its argument is wrapped as follows:

\[
\text{wrapFun} :: (\forall \alpha . \alpha \rightarrow \alpha) \rightarrow \text{Dynamic}
\]
\[
\text{wrapFun} f = \text{dynamic} f
\]

Analogously to unwrapping integers, a polymorphic function is unwrapped by specifying the expected type in a dynamic pattern match:

\[
\text{unwrapFun} :: \text{Dynamic} \rightarrow (\forall \alpha . \alpha \rightarrow \alpha)
\]
\[
\text{unwrapFun} (f :: \forall \alpha . \alpha \rightarrow \alpha) = f
\]
\[
\text{unwrapFun} _ = \bot
\]

The \(\alpha\) occurring in the type of the function is different from the same type variable in the dynamic pattern match; both have different binding sites.
Note that dynamic pattern matches can contain both type variables ($\alpha$, $\beta$, etc.) as well as pattern variables ($a$, $b$, etc.), the difference being that the former are explicitly bound by a universal quantifier while the latter are not. Consider the following function that tries to unwrap a function and apply it to a list:

$$
\begin{align*}
\text{dynAppList} ::& \forall \alpha . \text{Dynamic} \to [\alpha] \to \text{Dynamic} \\
\text{dynAppList} (f :: \forall \alpha . [\alpha] \to b) x &= \text{dynamic} (f x) \\
\text{dynAppList} \_ \_ &= \bot
\end{align*}
$$

The first arm only succeeds if the dynamic value contains a function that transforms any list regardless of the type of its elements, such as $\text{length} :: \forall \alpha . [\alpha] \to \text{Int}$, $\text{head} :: \forall \alpha . [\alpha] \to \alpha$, but also concatenation using $(+) :: \forall \alpha . [\alpha] \to [\alpha] \to [\alpha]$.

### 5.3.3 Type dependencies

The previous examples are context independent, in other words, the process of (un)wrapping values is not determined by the context in which these functions are used. Type dependencies allow the context to guide the (un)wrapping of values. A straightforward example is the following function that wraps any value in a dynamic value:

$$
\begin{align*}
\text{wrap} ::& \forall \alpha . \alpha \to \text{Dynamic} | \text{TC} \alpha \\
\text{wrap} x &= \text{dynamic} x
\end{align*}
$$

Here, the function is (ad-hoc) polymorphic in the argument type. We require the context to provide a so-called type code (i.e., the value representation of the type) using Clean’s built-in type class $\text{TC}$, which is stored together with the value. We elaborate later in Section 5.3.4 on this type class and type codes.

Similarly, we can unwrap values depending on the context:

$$
\begin{align*}
\text{unwrap} ::& \forall \alpha . \text{Dynamic} \to \alpha | \text{TC} \alpha \\
\text{unwrap} (x :: \alpha^\wedge) &= x \\
\text{unwrap} \_ \_ &= \bot
\end{align*}
$$

We require the value to be of the function result type by referring to the binding site of the same type variable using the $^\wedge$ annotation, omitting the universal quantifier from the dynamic pattern match. This causes the type code from the dynamic value to be unified with the type code obtained from the context, denoted by the $\text{TC}$ context. Therefore, success of the pattern match depends on the context in which the value is unwrapped.

As another example, we redefine $\text{dynApp}$ from Section 5.3.1 such that it depends on the context:

$$
\begin{align*}
\text{dynApp} ::& \forall \beta . \text{Dynamic} \to \text{Dynamic} \to \text{Maybe} \beta | \text{TC} \beta \\
\text{dynApp} (f :: a \to \beta^\wedge) (x :: a) &= f x \\
\text{dynApp} \_ \_ &= \bot
\end{align*}
$$

The function now only succeeds if the return type of the first argument fits the context, denoted by the type variable $\beta$ and the use of the $^\wedge$ annotation. Again, the $\text{TC}$ context is required so that the type code from the context is available to be compared to the type code stored in the dynamic value.
5.3.4 Type codes

As alluded to in the previous sections, type codes lie at the heart of dynamic typing; whenever a value is wrapped in a dynamic value, a type code is included as well. Also, functions with type dependencies require a type code to unify type information from the context with type information stored in a dynamic value. In Clean, a type code is provided by the built-in type class \( TC \):

\[
\text{class } TC \alpha \text{ where}
\]

\[
typeCode :: TypeCode
\]

It has a single member constant that provides a type code for its type argument. However, this type class is treated specially: any instance that is required is generated at compile time. Therefore, we cannot provide explicit instances of this type class. Type codes are defined by the vanilla datatype \( TypeCode \):

\[
:: TypeCode = \text{Scheme } [\text{String}] \text{ TypeCode}
\]

\[
| \text{Con TypeDef}
\]

\[
| \text{App TypeCode TypeCode}
\]

\[
| \text{Var String}
\]

A type code represents a universally quantified type with a list of variables, which is typically empty for monomorphic types. A type constructor is represented by a type definition, whose definition is left abstract for the sake of presentation. The definition of a type must be included since dynamic values can be (de)serialised across application boundaries. Then, verifying name equivalence during type checking simply does not suffice. Consequently, type codes cannot be defined for abstract types, but only for any nonabstract type. The other alternatives of \( TypeCode \) represent type application and type variables. As an example, we represent the type \( \forall \alpha . [\alpha] \to \text{Int} \) of the function \textit{length} by:

\[
\text{Scheme } ["a"]
\]

\[
(App (App (Con funDef)
\]

\[
(App (Con listDef)
\]

\[
(Var "a")))))
\]

\[
(Con intDef))
\]

The definitions of the function, list, and integer type codes are left abstract.

5.4 Dynamic typing in ad-hoc polymorphism

After the overview of both participants, we continue by considering the first side of their interaction: dynamic typing in the world of ad-hoc polymorphism. As a running example in this section, we consider applying the sorting function from Section 5.2.1 to a list that is unwrapped from a dynamic value. Naively, such a function would be defined as follows:
Since ad-hoc polymorphism is resolved at compile time, the challenge here to solve is that critical type information only becomes apparent at run time. Consequently, it is not known at compile time which instance must be provided to the sorting function. Alternatively, we could define a similar function that exposes the result type using type dependencies and the $\land$ annotation from Section 5.3.3:

$$
\text{dynSort} :: \forall \alpha . \text{Dynamic} \rightarrow [\alpha] \mid TC \alpha \\
\text{dynSort} \ (x :: [\alpha^\land]) = \text{sort} \ x \\
\text{dynSort} \ = = \bot
$$

Here, the dynamic type of the elements in the list is related to a static type. Though, ad-hoc polymorphism still cannot be resolved. The type code is provided as an abstract argument to this function, which cannot be used for resolving purposes at compile time.

A straightforward solution is to define this function without a pattern variable by enumerating all expected types:

$$
\begin{align*}
\text{dynSort} :: \text{Dynamic} \rightarrow \text{Dynamic} \\
\text{dynSort} \ (x :: \text{[Int]}) & = \text{dynamic} \ (\text{sort} \ x) \\
\text{dynSort} \ (x :: \text{[[Int]]}) & = \text{dynamic} \ (\text{sort} \ x) \\
\text{dynSort} \ (x :: \text{[[[Int]]]]) & = \text{dynamic} \ (\text{sort} \ x) \\
\text{dynSort} \ (x :: \text{[a]}) & = \text{dynamic} \ (\text{sort} \ x) \\
\text{dynSort} \ = & = \bot
\end{align*}
$$

This would resolve ad-hoc polymorphism at compile time since all required type information is provided manually. Evidently, this approach is cumbersome, prone to errors, and does not scale: we have to duplicate the right-hand side of the original function definition and easily forget an arm. Moreover, there are an infinite number of alternatives.

In this section, we specifically consider the situations where resolving ad-hoc polymorphism at compile time relies on type information that only becomes available at run time via pattern variables. We describe two complementary approaches to this challenge: container types (Section 5.4.1) and dynamic dictionary composition (Section 5.4.2).

### 5.4.1 Container datatypes

The first approach makes the producer of a dynamic value responsible for resolving ad-hoc polymorphism in future uses of this value. This is modelled by datatypes containing both values and their available instances, dubbed container datatypes; similar to classes in object-oriented languages. The most well-known form of container datatypes is existential datatypes (Läufer and Odersky, 1994). For example, the following datatype encapsulates a list value that can be ordered by the type of its elements:
:: $\text{EContOrdList} = \exists \alpha . \text{EContOrdList} ([\alpha] \mid \text{Ord} \alpha)$

The container prevents the type of its value from escaping, and only permits ordering operations. Since the notation for datatype alternatives coincides with Clean’s notation for type class contexts, we explicitly provide parentheses to denote that there must be an instance available for the type of the value, instead of a second alternative for the $\text{ContOrdList}$ type. A more permissive form of container datatypes is the following:

:: $\text{ContOrdList} \alpha = \text{ContOrdList} ([\alpha] \mid \text{Ord} \alpha)$

The existential type is pushed out of the definition such that the type of the value is exposed by the container. We often need such exposure to apply other operations than only the ones from the $\text{Ord}$ type class or to relate dynamic values to each other using pattern variables. Be aware that the semantics of such container datatypes is different from Haskell’s analogues definition:

```
data Ord $\alpha \Rightarrow \text{ContOrdList} \alpha = \text{ContOrdList} [\alpha]
```

Here, the context only guarantees that an instance exists, while Clean’s approach also makes the corresponding instance available when the constructor is pattern matched. Typically, the latter behaviour is achieved in Haskell using generalised algebraic datatypes (Peyton Jones et al., 2006):

```
data ContOrdList :: $\star \rightarrow \star$ where
    ContOrdList :: Ord $\alpha \Rightarrow [\alpha] \rightarrow \text{ContOrdList} [\alpha]
```

We use Clean’s container datatypes because generalised algebraic datatypes are not yet supported and we do not need its full power to model container datatypes.

**Semantics**

We define $\text{dynSort}$ again; now using the container datatype for $\text{Ord}$ on lists:

```
dynSort :: \text{Dynamic} \rightarrow \text{Dynamic}
dynSort (\text{ContOrdList} x :: \text{ContOrdList} \alpha) = \text{dynamic} (\text{sort} x)
dynSort _ = \bot
```

The dynamic pattern match is changed to include the (type) constructor of the container datatype. Note that we cannot use the existential variant here: its hidden element type escapes in that case to the type code included in the resulting dynamic value.

The context that is required by the use of $\text{sort}$ is statically provided by the local context that is propagated by pattern matching the container datatype. The semantics are very similar to context introduced by existential datatypes and generalised algebraic datatypes (Vytiniotis et al., 2011). Concretely, in dictionary-passing style, the container datatype for $\text{Ord}$ carries a dictionary in an extra field:

:: $\text{ContOrdList} \alpha = \text{ContOrdList} [\alpha] (\text{DictOrd} \alpha)$
Every construction of a container datatype fills in the appropriate dictionary, which
is accessed by the dynamic sorting function:

\[
\text{dynSort} :: \text{Dynamic} \rightarrow \text{Dynamic} \\
\text{dynSort} (\text{ContOrdList} \, x \, \text{da} :: \text{ContOrdList} \, a) = \text{dynamic} (\text{sort} \, \text{da} \, x) = \bot
\]

The obtained dictionary is simply passed on to the sorting function.

**Discussion**

The main advantage of container datatypes is that it is more a static approach
than a dynamic approach. It does require additional plumbing through (type)
constructors, but imposes a minimal run-time overhead. We specify the context
beforehand, allowing the corresponding dictionaries to be inserted at compile time.
The downside is that it requires us to predict all required contexts in advance,
something which can be hard. Therefore, this approach is better suited for ap-
lications that do not require much flexibility and are confined to strict interfaces.
For instance, when values are exchanged between applications and the permitted
operations need to be restricted.

A more worrying problem is that ambiguities quickly arise when multiple con-
tainer datatypes are used. For example, when we define a dynamic equality func-
tion. First, we define another container datatype that captures any value with its
Ord instance:

\[
:: \text{ContOrd} \, \alpha = \text{ContOrd} (\alpha \mid \text{Ord} \, \alpha)
\]

Then, we define the dynamic equality function as follows:

\[
\text{dynEq} :: \text{Dynamic} \rightarrow \text{Dynamic} \rightarrow \text{Bool} \\
\text{dynEq} (\text{ContOrd} \, x :: \text{ContOrd} \, a) (\text{ContOrd} \, y :: \text{ContOrd} \, a) = \text{eq} \, x \, y \\
\text{dynEq} = \bot
\]

Here, we statically enforce type equality of the two values by reusing the pattern
variable \(a\). We also require both values to be in a container together with their
Ord instance. However, there is no guarantee that the instance in the first value
is semantically equivalent to the instance in the second value. Possibly, these
dynamic values stem from different applications. It is only guaranteed that both
values have an instance available. Also, it is unspecified which one to choose. The
same issues arise when static contexts are mixed with dynamic contexts:

\[
\text{dynEq} :: \forall \, \alpha . \, \text{Dynamic} \rightarrow \alpha \rightarrow \text{Bool} \mid \text{Ord} \, \alpha \& \text{TC} \, \alpha \\
\text{dynEq} (\text{ContOrd} \, x :: \text{ContOrd} \, \alpha^\wedge) \, y = \text{eq} \, x \, y \\
\text{dynEq} = \bot
\]

Again, it is unclear whether to choose the instance for \(\text{Eq}\) obtained from the con-
tainer datatype or the context. We believe it is better to refuse such definitions
statically rather than to implement a complicated heuristic that solves the am-
biguities arbitrarily. A straightforward manual solution is to remove a container
datatype constructor using a helper function or a context from the function type,
depending on the desired behaviour.
5.4.2 Dynamic dictionary composition

In contrast to the first approach, the second approach makes the consumer of a dynamic value responsible for resolving ad-hoc polymorphism. In other words, the function that pattern matches a dynamic value has to come up with the appropriate instance. Since this depends on type information that becomes available at run time, we have to perform this process dynamically. Instead of translating the well-known static mechanism completely to their dynamic counterpart, we keep the quote of Meijer and Drayton from Section 5.1 in mind and only translate those parts across the dynamic border that cannot be performed statically. More concretely, only the composition of dictionary definitions needs to occur at run time.

Semantics

Again, we redefine \( \text{dynSort} \); now using an explicit context in the dynamic pattern match:

\[
\text{dynSort} :: \text{Dynamic} \rightarrow \text{Dynamic} \\
\text{dynSort} (x :: [a] | \text{Ord } a) = \text{dynamic } (\text{sort } x) \\
\text{dynSort } _\perp = \perp
\]

The required context introduced by the use of \text{sort} is now provided by the dynamic pattern match context. The pattern-match semantics of such contexts in a dynamic pattern match are straightforward: when there is no instance available at run time for the matched pattern variable, the pattern match fails. The requirement is in addition to the original dynamic pattern match semantics; the two-stage process is explicated in the translated dictionary-passing style definition of \text{dynSort}:

\[
\text{dynSort} :: \text{Dynamic} \rightarrow \text{Dynamic} \\
\text{dynSort} (x :: [a]) | gda = \text{dynamic } (\text{sort } da x) \\
\text{where} \\
(gda, da) = \text{guards } (\text{genDictOrd } (\text{dynamic } \perp :: a)) \\
\text{dynSort } _\perp = \perp
\]

The elegance of the translation is that this definition uses conventional dynamic typing mechanisms. In other words, the added context notation is merely syntactic sugar. The context is pushed out of the dynamic pattern match and turned into a guard that verifies the presence (i.e., creatability) of the required instance. The corresponding dictionary is obtained at run time by a generator function, given the type for which it has to compose one. This function groups the available instances and is mechanically constructed at compile time. We provide it the matched pattern variable as a value using a trick: we construct a dynamic value that is \( \perp \), relying on lazy evaluation, and explicitly provide an annotation that this value is of type \( a \). The resulting dictionary for this type is prepared by the helper function \text{guards} such that we can use the fall-through semantics of the guard:

\[
\text{guards} :: \forall \alpha . \text{Maybe Dynamic} \rightarrow (\text{Bool}, \alpha) | \text{TC } \alpha \\
\text{guards } (\text{Just } (x :: \alpha^\wedge)) = (\text{True}, x) \\
\text{guards } \text{Nothing} = (\text{False}, \perp)
\]
The second element of the resulting pair is only used when the first element passes a guard.

There is not necessarily a dictionary available for the type at hand. Also, because the type of each dictionary is different, the result of the generator function is wrapped in a dynamic value again. Hence, its type becomes:

\[ \text{genDictOrd} :: \text{Dynamic} \rightarrow \text{Maybe Dynamic} \]

Since this type permits almost any argument and result type, we have to keep correctness by construction in mind. The invariant we are guarding here is that given a value of type \( \alpha \), a dictionary of type \( \text{DictOrd} \alpha \) is returned, if there is one available. Typically, this is expressed through generalised algebraic datatypes, but Clean does not support such definitions as mentioned before in Section 5.4.1.

Recall from Section 5.2.1 that there are instances of \( \text{Ord} \) for integers, lists, and pairs. Each arm of the generator function follows mechanically from the available instances, each returning the corresponding dictionary from Figure 5.1. The first arm follows from the instance for integers:

\[
\text{genDictOrd} (\_ :: \text{Int}) = \text{Just } (\text{dynamic } \text{dictOrdInt})
\]

The corresponding \( \text{Ord} \) dictionary for integers is simply returned. The arm for lists requires a bit more work, using \( \text{unwrap} \) as defined in Section 5.3.3:

\[
\text{genDictOrd} (\_ :: [a]) = \text{do } da \leftarrow \text{genDictOrd} (\text{dynamic } \bot :: a) \\
\text{Just } (\text{dynamic } (\text{dictOrdList } (\text{unwrap } da)))
\]

As the instance header for lists dictates, a dictionary for \( \text{Ord} \) of the element type is required. Therefore, we match the type using a pattern variable and generate an \( \text{Ord} \) dictionary accordingly, borrowing Haskell’s \text{do}-notation for the sake of handling \( \text{Maybe} \) values conveniently. If this succeeds, we unwrap the result and construct the final dictionary for \( \text{Ord} \) of lists. Note that in general, unwrapping a dynamic value can fail. Here, we rely on correctness by construction as mentioned earlier. Similarly, the arm for pairs follows mechanically from its instance:

\[
\text{genDictOrd} (\_ :: (a, b)) = \text{do } da \leftarrow \text{genDictOrd} (\text{dynamic } \bot :: a) \\
\text{db} \leftarrow \text{genDictOrd} (\text{dynamic } \bot :: b) \\
\text{Just } (\text{dynamic } (\text{dictOrdPair } (\text{unwrap } da) (\text{unwrap } db)))
\]

We bind the different element types to pattern variables and generate \( \text{Ord} \) dictionaries for both element types. Then, if both result in a dictionary, we unwrap the results and generate a dictionary for \( \text{Ord} \) of pairs. Finally, a catch-all arm is defined if the presented type is none of the above (i.e., there is no instance available for this type):

\[
\text{genDictOrd } \_ = \text{Nothing}
\]

Note that though \( \text{Eq} \) is a superclass of \( \text{Ord} \), we do not have to consider its dictionary composition since these are already included in the available dictionaries for \( \text{Ord} \). The generator function merely composes these definitions as dictated by the available instances.
Discussion

Opposite to container datatypes, dynamic dictionary composition is more a dy-
namic approach; composition takes place at run time using a compile-time con-
structed generator function. Consequently, this approach is likely to intro-
duce more overhead at both compile time and run time. On the other hand, we are not
confronted with the additional plumbing of (type) constructors of container data-
types. But above all, we do not have to know the required contexts in advance;
operations are separated from values. Therefore, this approach is more suited to
applications that require more flexibility where fewer assumptions can be made
about the purpose of dynamically typed values. For example, when values are
obtained from user input.

Also, this approach does not suffer from the ambiguity problems that container
datatypes introduce. Consider a function for the equality of dynamic values, now
using dynamic dictionary composition. We include duplicate contexts on purpose
in the dynamic pattern matches:

\[
dynEq :: \text{Dynamic} \rightarrow \text{Dynamic} \rightarrow \text{Bool} \\
dynEq (x :: a \mid \text{Ord } a) (y :: a \mid \text{Ord } a) = eq x y \\
dynEq _ _ = \bot
\]

The crucial difference with container datatypes is that the contexts on Ord are
part of the dynamic pattern matches, not of the dynamic values themselves. The
same observation holds when the type of the dynamic value escapes to the context:

\[
dynEq :: \forall \alpha . \text{Dynamic} \rightarrow \alpha \rightarrow \text{Bool} \mid \text{Ord } \alpha \& \text{TC } \alpha \\
dynEq (x :: \alpha^\wedge \mid \text{Ord } \alpha^\wedge) y = eq x y \\
dynEq _ _ = \bot
\]

We always refer to the same type class and the same instances. Therefore, no
ambiguities can arise from duplicate contexts.

5.5 Ad-hoc polymorphism in dynamic typing

Now that we have seen how dynamic typing is included in the world of ad-hoc poly-
morphism, we continue by looking at the other way around: ad-hoc polymorphism
in the world of dynamic typing. We identify two challenges and consider the sort-
ing function from Section 5.2.1 as a running example in this section.

Naturally, the \textit{sort} function is wrapped as follows, using the \textit{wrap} function
from Section 5.3.3:

\[
\text{wrappedSort} :: \text{Dynamic} \\
\text{wrappedSort} = \text{wrap sort}
\]

The first challenge is to come up with an appropriate type code, as dictated by
the type of \textit{wrap}.

One of the possibilities to unwrap values from dynamic values is by using the
\textit{unwrap} function from Section 5.3.3. For example, a function that unwraps and
applies a value like \textit{wrappedSort} is naively defined as follows:
5.5 Ad-hoc polymorphism in dynamic typing

\[ \text{dynAppOrd} :: \forall \alpha . \text{Dynamic} \rightarrow [\alpha] \rightarrow [\alpha] \mid \text{Ord} \alpha \]
\[ \text{dynAppOrd} \ d \ x = \text{unwrap} \ d \ x \]

Unfortunately, this definition does not capture the intended behaviour. The type inferred for the result of unwrapping the value is too general; namely \( \forall \alpha . [\alpha] \rightarrow [\alpha] \). Since we explicitly require an ad-hoc polymorphic function, we have to provide an explicit type signature as well. In general, unwrapping ad-hoc polymorphic values always requires an explicit dynamic pattern match:

\[ \text{dynAppOrd} :: \forall \alpha . \text{Dynamic} \rightarrow [\alpha] \rightarrow [\alpha] \mid \text{Ord} \alpha \]
\[ \text{dynAppOrd} \ (f :: \forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord} \alpha) \ x = f \ x \]
\[ \text{dynAppOrd} \ _ = \bot \]

This function is ad-hoc polymorphic of its own; it has to propagate the \text{Ord} context to the unwrapped function. The first arm now includes an explicit dynamic pattern match that specifies an ad-hoc polymorphic type. The syntactic difference with the function \text{dynSort} from Section 5.4.2 is subtle: there we used a pattern variable while here we use a universally quantified type variable. The semantics of the former is opposite to the latter: there we have to produce an instance, while here we consume an instance. Here, the challenge is to extend existing semantics for unwrapping ad-hoc polymorphic values.

In this section we describe two different approaches to these challenges: dictionary-passing types (Section 5.5.1) and type code extension (Section 5.5.2).

### 5.5.1 Dictionary-passing types

The first approach makes clever use of the dictionary-passing style translation of type classes. This translation ‘removes’ ad-hoc polymorphism from a type, obtaining an ordinary (i.e., parametric) polymorphic type. Consider the dictionary-passing type of the function \text{sort}, as given before in Section 5.2.2:

\[ \text{sort} :: \forall \alpha . \text{DictOrd} \alpha \rightarrow [\alpha] \rightarrow [\alpha] \]

When wrapping such a value in a dynamic value, a type code is required for its ad-hoc polymorphic type. However, we can make use of the existing type code mechanisms by requiring a type code for its dictionary-passing type instead.

Analogously, this approach translates dynamic pattern matches with an ad-hoc polymorphic type to a dictionary-passing type as well. For instance, the \text{dynAppOrd} function becomes:

\[ \text{dynAppOrd} :: \forall \alpha . \text{DictOrd} \alpha \rightarrow \text{Dynamic} \rightarrow [\alpha] \rightarrow [\alpha] \]
\[ \text{dynAppOrd} \ da \ (f :: \forall \alpha . \text{DictOrd} \alpha \rightarrow [\alpha] \rightarrow [\alpha]) \ x = f \ da \ x \]
\[ \text{dynAppOrd} \ _ = \bot \]

Since the type in the dynamic pattern match is no longer ad-hoc polymorphic, it only succeeds if it is provided a value that is exactly ad-hoc polymorphic in \text{Ord}: nothing more, nothing less.
5 Ad-hoc polymorphism and dynamic typing

Discussion

The advantage of the first approach is that it is lightweight: it is defined in terms of existing mechanisms. Consequently, the semantics of unwrapping ad-hoc polymorphic values does not take superclass relations into account. Also, this approach offers a backdoor. Since it translates dynamic pattern matches to include a dictionary-passing type, we can obtain the dictionary that is usually kept hidden from us. For example, when the expression `dynamic (λda x → x)` is presented as the first argument of `dynAppOrd`, the dynamic pattern match succeeds and the variable `da` is bound to the hidden internal dictionary. Although we cannot use the value since its dictionary type remains hidden internally, it is not very elegant.

5.5.2 Type code extension

The second approach relies less on existing mechanisms and extends these where necessary. As mentioned before, the use of the function `wrap` in the example function `wrappedSort` requires a type code for the ad-hoc polymorphic type of `sort`. The described type code definition in Section 5.3.4 extends naturally to include such types. Recall that we only consider type classes in the style of Haskell 98, where contexts in type signatures are of the form `C α` or `C (α τ_1 . . . τ_n)`, where `C` is a type class, `α` a type variable, and `τ_i` is any type. Then, we add a list of contexts to the `Scheme` alternative of the `TypeCode` type:

```
:: TypeCode = Scheme [String] TypeCode [Context] 
  | ... 
:: Context = Context ClassDef Parameter 
:: Parameter = Parameter String [Type] 
:: ClassDef 
:: Type
```

Similar to type constructors, a context includes a type class definition since name equivalence does not suffice. Also it contains a parameter, which is defined by a variable and a list of types. The list is empty for contexts of the form `C α`. The definition of type class definitions and types is left abstract. Then, the ad-hoc polymorphic type of sort `∀ α . [α] → [α] | Ord α` is represented as follows:

```
Scheme ["a"]
  (App (App (Con funDef)
         (App (Con listDef)
               (Var "a")))))
  (App (Con listDef)
       (Var "a")))
[Context ordDef (Parameter "a" [])]
```

We leave the definition of the ordering type class, as well as the function and list type, abstract.

Since this approach extends the existing type code definitions, we have to extend the semantics for dynamic pattern matches involving such type codes as
well. The proposed semantics as described before in Section 5.5.1 is unnecessarily restrictive. To illustrate this, we make a brief excursion to a similar phenomenon and consider rank-2 polymorphism (Odersky and Läufer, 1996; Peyton Jones et al., 2007). Suppose we define the rank-2 analogue of \textit{dynAppOrd} as follows:

\[
\begin{align*}
\text{rank2AppOrd} &:: \forall \alpha . (\forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha) \rightarrow [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha \\
\text{rank2AppOrd } f \ x & = f\ x
\end{align*}
\]

Here, we choose to lift the type from the first dynamic pattern match to the type of the function, replacing the occurrence of \textit{Dynamic}. The first argument of the function is ad-hoc polymorphic and still expects a dictionary, which is provided by the context of \textit{rank2AppOrd}. Evidently, we can omit the default case safely because the function no longer operates on dynamic values but on one specific lifted type. To gain insight in the desired semantics of dynamic pattern matches with ad-hoc polymorphic types, we look at possible arguments to \textit{rank2AppOrd}. From less general to more general, we consider functions for sorting, removing duplicates from, and reversing lists:

\[
\begin{align*}
\text{sort} &:: \forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha \\
\text{nub} &:: \forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Eq } \alpha \\
\text{reverse} &:: \forall \alpha . [\alpha] \rightarrow [\alpha]
\end{align*}
\]

Clearly, providing the sorting function to \textit{rank2AppOrd} is well typed; their types precisely match. Perhaps surprisingly, the duplicate removal function is suited as well; its type is more general than the sorting function since \textit{Eq} is a superclass of \textit{Ord}. Similarly, the reversing function poses no problem with the most general type of the three; it contains no contexts at all. Ideally, dynamic pattern matches with ad-hoc polymorphic types exhibit the same semantics. Then, the first arm of \textit{dynAppOrd} must succeed for \textit{sort}, \textit{nub}, and \textit{reverse} as well.

**Discussion**

Opposite to the first approach, this approach is more heavyweight: it requires an extension of type codes and includes rank-2 polymorphism semantics in type checking. On the other hand, this approach does not provide any backdoors and is therefore more elegant. Moreover, it is more flexible in dynamic pattern matches. However, it can be desirable to precisely pattern match on contexts, not taking superclass relations into account. Luckily, such behaviour is achieved by enumerating the cases from more general to less general. For example, we distinguish the three function types of \textit{sort}, \textit{nub}, and \textit{reverse} by the following ordering of dynamic pattern matches:

\[
\begin{align*}
\text{distinguish} :: \text{Dynamic} \rightarrow \ldots \\
\text{distinguish } (f :: \forall \alpha . [\alpha] \rightarrow [\alpha]) & = \ldots \\
\text{distinguish } (f :: \forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Eq } \alpha) & = \ldots \\
\text{distinguish } (f :: \forall \alpha . [\alpha] \rightarrow [\alpha] \mid \text{Ord } \alpha) & = \ldots \\
\text{distinguish } & = \bot
\end{align*}
\]

The first arm matches only \textit{reverse}, the second arm matches \textit{nub} as well, and the final arm matches all three functions. Consequently, if we wish to distinguish all \(n\) superclasses of a type class, this results in \(n + 1\) arms.
5.6 Type class extensions

Until now, we have only considered the realisation of ad-hoc polymorphism through type classes in the style of Haskell 98. In this section we discuss some of the more popular type class extensions (Peyton Jones et al., 1997) in Clean and Haskell: multi-parameter type classes (Section 5.6.1), flexible contexts (Section 5.6.2), and flexible instances (Section 5.6.3). We give a brief introduction to each extension and discuss if and how it affects dynamic typing in ad-hoc polymorphism, as in Section 5.4, and ad-hoc polymorphism in dynamic typing, as in Section 5.5. Not all of the described approaches are affected by all extensions though. Table 5.1 provides additional guidance for this section by summarising the interactions that require discussion.

5.6.1 Multi-parameter type classes

We assumed in earlier sections that the number of type class parameters is restricted to one. The multi-parameter type class extension lifts the restriction so that a type class can have any number of parameters, which do not necessarily need to be distinct variables. Consider the following multi-parameter type class, one that models an array of type $\alpha$ with elements of type $\epsilon$, with a single member function that returns the value at the indicated position:

\[
\text{class } \text{Array } \alpha \text{ } \epsilon \text{ where}
\text{select :: Int } \rightarrow \alpha \text{ } \epsilon \rightarrow \epsilon
\]

Then, an instance for lists of integers is defined as follows:

\[
\text{instance } \text{Array } [\text{Int} \text{ where}}
\text{select } i x = \ldots
\]

The corresponding dictionary type now takes two parameters:

\[
\text{:: DictArray } \alpha \text{ } \epsilon = \{ \text{select :: Int } \rightarrow \alpha \text{ } \epsilon \rightarrow \epsilon \}
\text{dictArrayListInt :: DictArray } [\text{Int} \text{ where}}
\text{dictArrayListInt } = \{ \text{select } = \lambda i x \rightarrow \ldots \}
\]

Evidently, also the form of contexts occurring in type signatures change accordingly. For example in the function that selects the first element of an array:
5.6 Type class extensions

\[ \text{selectFirst} :: \forall \alpha \epsilon . \alpha \epsilon \rightarrow \epsilon \mid \text{Array} \alpha \epsilon \]
\[ \text{selectFirst} = \text{select} \ 0 \]

Since the complete form of type classes is affected by this extension, dynamic typing in ad-hoc polymorphism as well as ad-hoc polymorphism in dynamic typing is affected.

**Dynamic typing in ad-hoc polymorphism**

Similar to \textit{dynSort} as defined in Section 5.4, we naively define a dynamic function that selects the first element of an array as follows:

\[ \text{dynSelectFirst} :: \text{Dynamic} \rightarrow \text{Dynamic} \]
\[ \text{dynSelectFirst} \ (x :: a \ e) = \text{dynamic} \ (\text{selectFirst} \ x) \]
\[ \text{dynSelectFirst} \ _ = \bot \]

Again, type information that is required to resolve ad-hoc polymorphism only becomes available at run time. We describe how both container datatypes and dynamic dictionary composition are extended to support resolving of multi-parameter type classes.

**Container datatypes** The extension is incorporated naturally in container datatypes. For instance, when we define a container that captures values together with their \texttt{Array} instance:

\[ :: \text{ContArray} \alpha \epsilon = \text{ContArray} ((\alpha \epsilon) \mid \text{Array} \alpha \epsilon) \]

The container datatype now takes two parameters, in contrast to the similar definition of \texttt{ContOrd} from Section 5.4.1. Then, we adapt the function to include the constructor of the container datatype:

\[ \text{dynSelectFirst} :: \text{Dynamic} \rightarrow \text{Dynamic} \]
\[ \text{dynSelectFirst} \ (\text{ContArray} \ x :: \text{ContArray} \ a \ e) = \text{dynamic} \ (\text{selectFirst} \ x) \]
\[ \text{dynSelectFirst} \ _ = \bot \]

As before, the container datatype carries an additional field in dictionary-passing style:

\[ :: \text{ContArray} \alpha \epsilon = \text{ContArray} (\alpha \epsilon) \ (\text{DictArray} \alpha \epsilon) \]

Then, the \textit{dynSelectFirst} function makes the dictionary available in the dynamic pattern match:

\[ \text{dynSelectFirst} :: \text{Dynamic} \rightarrow \text{Dynamic} \]
\[ \text{dynSelectFirst} \ (\text{ContArray} \ x \ da :: \text{ContArray} \ a \ e) = \]
\[ \quad \text{dynamic} \ (\text{selectFirst} \ da \ x) \]
\[ \text{dynSelectFirst} \ _ = \bot \]

The obtained dictionary is passed on to the \textit{selectFirst} function.
Dynamic dictionary composition. Supporting the extension in dynamic dictionary composition requires just a bit more work. As an example, consider dynSelectFirst where an explicit context is included in the dynamic pattern match:

\[
\text{dynSelectFirst} :: \text{Dynamic} \rightarrow \text{Dynamic} \\
\text{dynSelectFirst} (x :: a \; e \mid \text{Array} \; a \; e) = \text{dynamic} \; (\text{selectFirst} \; x) \\
\text{dynSelectFirst} _- = \perp
\]

As before, this definition translates mechanically to the following:

\[
\text{dynSelectFirst} :: \text{Dynamic} \rightarrow \text{Dynamic} \\
\text{dynSelectFirst} (x :: a \; e) \mid \text{gda} = \text{dynamic} \; (\text{selectFirst} \; da \; x) \\
\text{where} \\
(\text{gda}, \text{da}) = \text{guards} \; (\text{genDictArray} \; (\text{dynamic} \; \perp :: a) \; (\text{dynamic} \; \perp :: e)) \\
\text{dynSelectFirst} _- = \perp
\]

Now, the generator function for \text{Array} takes two arguments, one for each of its parameters. Note that the first parameter \(a\) of the generator function is of kind \(\star \rightarrow \star\). However, Clean requires such a type to be of kind \(\star\). This is easily solved by fully saturating the pattern variable with type variables that are universally quantified locally, giving us the type \(\forall \alpha . \; a \; \alpha\). For the sake of presentation, we will not saturate higher-kinded argument types of the generator function explicitly. Due to the two parameters, the type of the generator function becomes:

\[
\text{genDictArray} :: \text{Dynamic} \rightarrow \text{Dynamic} \rightarrow \text{Maybe} \; \text{Dynamic}
\]

The instance of \text{Array} for lists and integers dictates the following arm that includes a dynamic pattern match for both parameters:

\[
\text{genDictArray} (_- :: [[]]) (_- :: \text{Int}) = \text{Just} \; (\text{dynamic} \; \text{dictArrayListInt})
\]

Similar to before, the first dynamic pattern match contains a type of kind \(\star \rightarrow \star\), while Clean requires it to be of kind \(\star\). This is solved in the same fashion by fully saturating the type in the dynamic pattern match with type variables, obtaining \(\forall \alpha . \; [\alpha]\) of the proper kind. Again, for the sake of presentation, we will not saturate higher-kinded types in dynamic pattern matches explicitly.

Now that contexts consist of multiple parameters, some of the types can be known statically while others still depend on dynamic type information. For example, when the type of the elements in the array is dictated by the context in which the dynSelectFirst function is used:

\[
\text{dynSelectFirst} :: \forall \epsilon . \; \text{Dynamic} \rightarrow \epsilon \mid \text{TC} \; \epsilon \\
\text{dynSelectFirst} (x :: a \; e^\wedge \mid \text{Array} \; a \; e^\wedge) = \text{selectFirst} \; x \\
\text{dynSelectFirst} _- = \perp
\]

Maybe surprisingly, the illustrated translation still holds. The generator function is still provided two arguments, where the first is of type \(a\) but where the second is now of type \(e^\wedge\); type information from the \(^\wedge\) annotation is propagated to the generator function.
Ad-hoc polymorphism in dynamic typing

Wrapping a value like `selectFirst` remains straightforward:

\[
\begin{align*}
\text{wrappedSelectFirst} &:: \text{Dynamic} \\
\text{wrappedSelectFirst} &= \text{wrap selectFirst}
\end{align*}
\]

Unwrapping and applying such a value consists of a dynamic pattern match with the corresponding ad-hoc polymorphic type:

\[
\begin{align*}
\text{dynAppArray} &:: \forall \alpha \epsilon . \text{Dynamic} \rightarrow \alpha \epsilon \rightarrow \epsilon | \text{Array} \alpha \epsilon \\
\text{dynAppArray} (f :: \forall \alpha \epsilon . \alpha \epsilon \rightarrow \epsilon | \text{Array} \alpha \epsilon) x &= f x \\
\text{dynAppArray} _- &= \perp
\end{align*}
\]

Since dictionary-passing types and type code extension rely on the form of type classes, these have to take the extension into account.

Dictionary-passing types  The dictionary-passing types approach is straightforwardly extended. When a value like `selectFirst` is wrapped in a dynamic value, a type code for its dictionary-passing type is included in the dynamic:

\[
\begin{align*}
\text{selectFirst} &:: \forall \alpha \epsilon . \text{DictArray} \alpha \epsilon \rightarrow \alpha \epsilon \rightarrow \epsilon
\end{align*}
\]

Fortunately, type codes already support type constructors applied to multiple arguments. Also, unwrapping such a value via a pattern match in the `dynAppArray` function translates naturally to include a dictionary-passing type:

\[
\begin{align*}
\text{dynAppArray} &:: \forall \alpha \epsilon . \text{DictArray} \alpha \epsilon \rightarrow \text{Dynamic} \rightarrow \alpha \epsilon \rightarrow \epsilon \\
\text{dynAppArray} da (f :: \forall \alpha \epsilon . \text{DictArray} \alpha \epsilon \rightarrow \alpha \epsilon \rightarrow \epsilon) x &= f da x \\
\text{dynAppArray} da _- &= \perp
\end{align*}
\]

The semantics for the dynamic pattern match remains the same: it only succeeds if it is provided a value that is exactly ad-hoc polymorphic in `Array`.

Type code extension  Wrapping ad-hoc polymorphic values like `selectFirst` requires a modification to the type codes presented earlier in Section 5.5.2. Luckily, these are easily extended with multiple parameters by adapting the `Context` type accordingly:

\[:: \text{Context} = \text{Context ClassDef [Parameter]}\]

Instead of a single parameter, a context now includes a list of parameters. Also, the `ClassDef` has to take the new form into account. Unwrapping a value like `selectFirst` still follows the semantics of the analogue rank-2 polymorphic definition.

Flexible contexts

We restricted contexts occurring in type class definitions, instance definitions, and type signatures as well. In type class definitions, contexts are restricted to the form
C α, where C is a type class and α a type variable. The flexible contexts extension allows us to define any context (i.e., superclass) in a type class definition, as long as the class hierarchy remains acyclic. This change only affects dictionary types and the translation from concrete instances to dictionary definitions. Therefore, this is not of our concern. However, flexible contexts of defined instances do affect our approach. In combination with multi-parameter type classes as described earlier, it relaxes the original context of the form C α to be C τ1 . . . τn, where τi is any type. To ensure compile-time termination, the new form is subject to the so-called Paterson and Coverage conditions (Sulzmann et al., 2007). As an example of flexible contexts, we define an instance of Array for lists and lists of values:

```haskell
instance Array [] [ε] | Array [] ε where
    select i x = ...
```

Now, the context includes the type constructor []. Its dictionary definition takes an additional argument that reflects this context:

```haskell
dictArrayListList :: ∀ ε . DictArray [] ε → DictArray [] [ε]
dictArrayListList da = \{ select = λi x → ... \}
```

The final part of the extension lifts the restriction of the form of contexts in type signatures from C α or C (α τ1 . . . τn) to C τ1 . . . τn. For instance, in the following example we explicitly require a list array to concatenate the nested lists:

```haskell
concatArray :: ∀ ε . [[ε]] → [ε] | Array [] ε
```

We leave its definition abstract since this requires a more elaborate Array type class with more member functions.

Both dynamic typing in ad-hoc polymorphism as well as ad-hoc polymorphism in dynamic typing are affected by flexible contexts. The former since the form of contexts in instances is changed and the latter since the form of contexts in type signatures is changed.

**Dynamic typing in ad-hoc polymorphism**

We only have to consider dynamic dictionary composition since container data types are not concerned with more flexible contexts in type class definitions, instance definitions, or type signatures.

**Dynamic dictionary composition**  Consider the earlier introduced generator function for the Array type class from Section 5.6.1. The instance for lists and lists of values, defined using the flexible contexts extension, adds the following arm:

```haskell
genDictArray (⊥ :: []) (⊥ :: [ε]) =
    do da ← genDictArray (dynamic ⊥ :: []) (dynamic ⊥ :: e)
       Just (dynamic (dictArrayListList (unwrap da))
```

Here, the recursive call to the generator function no longer just takes pattern variables obtained from the dynamic pattern match, but ordinary types as well,
as visible in its first argument. The expressive dynamic type system allows us to construct a dynamic value of any type, therefore, this extension is easily taken care of. Note that if we lifted the Paterson and Coverage conditions mentioned earlier, termination of the generator function is not guaranteed.

### Ad-hoc polymorphism in dynamic typing

A value like `concatArray` is wrapped as usual:

\[
\text{wrappedConcatArray} :: \text{Dynamic} \\
\text{wrappedConcatArray} = \text{wrap concatArray}
\]

Again, such a value is unwrapped by explicating its type in a dynamic pattern match:

\[
\text{dynAppArray} :: \forall \epsilon . \text{Dynamic} \rightarrow [[\epsilon]] \rightarrow [\epsilon] | \text{Array} [] \epsilon \\
\text{dynAppArray} \ (f :: \forall \epsilon . [[\epsilon]] \rightarrow [\epsilon] | \text{Array} [] \epsilon) \ x = f \ x \\
\text{dynAppArray} \ _ = \bot
\]

Since the form of contexts in type signatures is changed by the extension, both dictionary-passing types and type code extension are affected.

### Dictionary-passing types

Flexible contexts are straightforwardly included in the dictionary-passing types approach. For instance, when we wrap the function `concatArray`, a type code is included for the following type:

\[
\text{concatArray} :: \forall \epsilon . \text{DictArray} [] \epsilon \rightarrow [[\epsilon]] \rightarrow [\epsilon]
\]

As before, the dictionary-passing type is used in the dynamic pattern match as well:

\[
\text{dynAppArray} :: \forall \epsilon . \text{DictArray} [] \epsilon \rightarrow \text{Dynamic} \rightarrow [[\epsilon]] \rightarrow [\epsilon] \\
\text{dynAppArray} \ da \ (f :: \forall \epsilon . \text{DictArray} [] \epsilon \rightarrow [[\epsilon]] \rightarrow [\epsilon]) \ x = f \ da \ x \\
\text{dynAppArray} \ da \ _ = \bot
\]

Note that the dynamic pattern match not only succeeds any more for values that are exactly ad-hoc polymorphic in `Array`. Since the first parameter of the dictionary type is restricted to the list type, any dictionary that is less restrictive will also do.

### Type code extension

We adapt the `Parameter` type, that models type class parameters, from Section 5.5.2 to include flexible contexts:

\[
:: \text{Parameter} = \text{Parameter} [\text{Type}]
\]

It now includes a list of types, instead of always requiring a type variable in prefix position. Again, the `ClassDef` type has to be modified as well to include the new form of contexts in type class definitions. The unwrapping of values including flexible contexts follows rank-2 polymorphism semantics.
5.6.3 Flexible instances

Besides a restricted context, we also restricted the instances to be of the form $C (T \alpha_1 \ldots \alpha_n)$ where $C$ is a type class, $T$ a type constructor, and $\alpha_i$ a distinct type variable. The flexible instances extension lifts this restriction, including multi-parameter type classes, to the form $C \tau_1 \ldots \tau_n$, where $C$ is a type class and $\tau_i$ is any type. For example, we define an instance of $Array$ for lists and pairs with integers:

```haskell
instance Array [] (Int, e) where
  select i x = …
```

The corresponding dictionary definition reflects the flexible instance in its type:

```haskell
dictArrayListPairInt :: ∀ e . DictArray [] (Int, e)
dictArrayListPairInt = { select = λ i x → … }
```

Due to the extension, overlap between instances can occur. For example, consider the following additional instance:

```haskell
instance Array α (Int, e) | Array α e where
  select i x = …
```

Here, the instances overlap in the first parameter of $Array$. Consequently, it is not clear which instance to choose for an array whose elements are of type $(Int, Bool)$. Normally, such ambiguities are rejected at compile time. Another extension called overlapping instances lifts this restriction and chooses the most specific one. In this example, the first one is most specific. Evidently, there is not always such an instance, consider for example the following:

```haskell
instance Array [] (e, Bool) where
  select i x = …
```

Then, the overlapping instances are rejected at compile time.

Since the extension only affects the form of instances, only dynamic typing in ad-hoc polymorphism is affected.

Dynamic typing in ad-hoc polymorphism

Only dynamic dictionary composition is concerned with the form of instances. Therefore, we do not consider container datatypes.

Dynamic dictionary composition

As before, each instance of a type class results in an arm of the corresponding generator function. The instance of $Array$ for lists and pairs with integers gives the following arm:

```haskell
genDictArray (λ :: []) (λ :: (Int, e)) = Just (dynamic dictArrayListPairInt)
```

The more flexible form of the second parameter results in a more elaborate dynamic pattern match. Again, the dynamic type system is expressive enough to cope with such types in a dynamic pattern match.
Overlapping instances are resolved at compile time. If there is not a single most specific instance to choose, a compile-time error occurs. In our approach we use a generator function to compose dictionaries. Therefore, it is only until run time that we are able to verify this condition. Unfortunately, the generator function is mechanically constructed at compile time, choosing an explicit ordering of the arms. Consequently, overlapping instances are not supported in this approach.

5.7 Related work

An extensive overview of the interaction between ad-hoc polymorphism and dynamic typing in a statically typed functional language has not been described earlier. However, bringing the worlds of ad-hoc polymorphism and dynamic typing has been recognised before (Plasmeijer and Van Weelden, 2005). An interactive shell is described to interpret user-provided values using the dynamic type system. To facilitate ad-hoc polymorphism, the dictionary-passing style is made explicit when translating the value provided by the user to an internal structure. Unfortunately, this is restricted to predefined type classes; additional instances cannot be provided by the user. Evidently, the approaches described in this chapter are not restricted in that sense. Furthermore, our approach is more flexible since we are not confined to the world of dynamic typing.

Before type classes, Kaes (1988) already described an approach towards ad-hoc polymorphism named parametric overloading where functions are parameterised with additional arguments that capture the abstracted behaviour. In that sense, these functions are not ad-hoc polymorphic but parametric polymorphic. Wadler and Blott (1989) improved on this technique by allowing the additional parameters to be grouped in type classes, and described the translation to dictionary-passing style which we used extensively in this chapter. However, the approach of parametric overloading did include a mechanism that resolves ad-hoc polymorphism dynamically. Unfortunately, this requires the additional parameters (i.e., the dictionary) to be strict, and possibly resulted in nontermination. Our approaches of container datatypes and dynamic dictionary composition do not use parametric overloading but include the full power of type classes without compromising laziness nor termination.

Leroy and Mauny (1993) describe dynamic typing with polymorphism and show how this is used to model ad-hoc polymorphism. Functions enumerate all possible expected types using dynamic values, which is named structural ad-hoc polymorphism. Its opposite is embodied by type classes and is called nominal ad-hoc polymorphism. The main difference is that the nominal variant is ‘open’ (i.e., instances can be given anywhere), whereas the structural variant is ‘closed’ (i.e., ad-hoc polymorphic functions enumerate the possible cases). The former is orthogonal to the latter: instances do not require exhaustive enumerations, though their definitions are dispersed. A unification of both in a single functional language is described by Vytiniotis et al. (2004). Our dynamic dictionary composition approach uses both: dispersed instances are mechanically grouped at compile time in a single generator function to capture all available instances.

While we considered statically typed functional languages like Clean, Haskell, and ML, other functional languages that are dynamically typed also support ad-
ad-hoc polymorphism. For instance, languages like Lisp and Scheme resolve ad-hoc polymorphism at run time since only then type information becomes available. While these languages use a similar dispatching mechanism like the generator function, there is no support for an expressive static system like type classes.

We describe run-time resolving of ad-hoc polymorphism that can fail, consistent with the original semantics of dynamic pattern matches. Rouaix (1990) describes an approach, inspired by object-oriented languages, where a restricted form of ad-hoc polymorphism is resolved at run time without any possibility of run-time failure. However, this is described in a statically typed language, while our approaches especially consider languages that support dynamic typing as well.

Object-oriented languages, being statically typed like Java and Scala or dynamically typed like Smalltalk, resort to run-time resolving of ad-hoc polymorphism due to their late binding. Only at run time it can be determined which method is used.

5.8 Conclusion

We have given an elaborate overview of the interaction between ad-hoc polymorphism and dynamic typing in a statically typed functional language. We identified two sides to their interaction: dynamic typing in ad-hoc polymorphism and ad-hoc polymorphism in dynamic typing, introducing one world into the other. Regarding the former interaction, we showed two complementary approaches, namely container datatypes and dynamic dictionary composition, that provide mechanisms to resolve ad-hoc polymorphism depending on dynamic type information. Both approaches are best suited in different applications, either requiring rigidity or flexibility. Also, both approaches can happily coexist. With respect to the latter interaction, we showed two different approaches, namely dictionary-passing types and type code extension, to wrap and unwrap ad-hoc polymorphic values. These approaches differ in implementation effort and flexibility of pattern-matching semantics. Finally, we discussed several type class extensions and argued that most of these fit naturally in the described mechanisms. Only lifting the restrictions of flexible contexts using undecidable instances and flexible instances using overlapping instances are not supported by dynamic dictionary composition.

Some of the work described in this chapter has been experimentally included in Clean: container datatypes that expose the type of their content, as well as the possibility to (un)wrap ad-hoc polymorphic values via dictionary-passing types. We plan to experiment with the other approaches in Clean as well. Also, we aim to further investigate the relation between ad-hoc polymorphism and dynamic typing via a more formal approach to their interactions.
6 Embedding polymorphic dynamic typing

Abstract
Dynamic typing in a statically typed functional language allows us to defer type checking until run time. This is typically useful when interacting with the ‘outside’ world where the type of values involved may not be known statically. Haskell has minimal support for dynamic typing: it only supports monomorphism. Clean, on the other hand, has a more rich and mature dynamic type system where polymorphism is supported as well. An interesting difference is that Haskell offers monomorphic dynamic typing via a library, while Clean offers polymorphic dynamic typing via built-in language support. The advantage of this approach is that it is defined on abstract syntax trees, whereas a library is restricted by the expressivity of the language itself. On the other hand, the Haskell approach does not need to extend the core language and hence reduces the complexity of the language and compiler. In this chapter we investigate what it takes for a functional language to embed polymorphic dynamic typing. We explore such an embedding in Haskell using generalised algebraic datatypes and argue that a universe for the representation of types needs to be separated from its interpretation as a type. We motivate the need for a dependently-typed language like Agda and perform the embedding using structural equality on type representations. Then, we extend this approach with an instance-of algorithm and give a complete proof of its correctness in Agda itself. Finally, we define the corresponding cast function.

6.1 Introduction
Dynamic typing in a statically typed functional language such as Clean and Haskell allows us to defer type checking until run time. This is typically useful when interacting with the ‘outside’ world: when values are exchanged between applications by deserialisation from disk, input is obtained from a user, or when values are obtained via a network connection. In such situations, the types of the values at hand may not be known until run time. Values and functions are wrapped together with a representation of their type in a uniform black box, as their type is statically not known. Such a value is unwrapped by pattern matching and specifying the expected type. Although type checking can fail at run time when a dynamic value presents an unexpected type, the static type system guarantees that when pattern matching succeeds, the unwrapped value can be used in a type-safe fashion from there on. Hence, the advantages of static typing are not compromised.
Haskell has minimal support for dynamic typing: it only supports monomorphism (Baars and Swierstra, 2002; Cheney and Hinze, 2002). The flagship Haskell compiler GHC includes a library function `toDyn` to wrap any monomorphic value in a dynamic value. Consequently, wrapping a polymorphic value requires us to give the value a monomorphic type explicitly. For example, consider wrapping the polymorphic identity function in a dynamic value:

\[
\text{idDyn :: Dynamic} \\
\text{idDyn = toDyn ((\lambda x \rightarrow x) :: Int \rightarrow Int)}
\]

Then, a value is unwrapped using the library function `fromDyn` which performs a cast, where the expected type is specified by the context in which it is unwrapped:

\[
\text{idInt :: Maybe (Int \rightarrow Int)} \\
\text{idInt = fromDyn idDyn}
\]

Clean, on the other hand, has a more rich and mature dynamic type system that is built-in; it adopted ML’s support for monomorphism (Abadi et al., 1991) and polymorphism (Leroy and Mauny, 1993; Abadi et al., 1995; Pil, 1997). Having such an extensive dynamic type system does not only improve orthogonality with the static type system, as discussed in Chapter 5, but also has important applications (Plasmeijer and Van Weelden, 2005; Plasmeijer et al., 2011). In Clean, we wrap a value, such as the polymorphic identity function, in a dynamic value using the keyword `dynamic`:

\[
\text{idDyn :: Dynamic} \\
\text{idDyn = \text{dynamic (\lambda x \rightarrow x) :: } \forall \alpha . \alpha \rightarrow \alpha}
\]

Then, we unwrap such a value by pattern matching and specifying the expected type using the :: annotation:

\[
\text{id :: Maybe (\forall \alpha . \alpha \rightarrow \alpha)} \\
\text{id = case idDyn of} \\
\text{ (f :: \forall \alpha . \alpha \rightarrow \alpha) \rightarrow Just f} \\
\text{ - } \rightarrow \text{Nothing}
\]

It is important to observe that the expected type does not need to be structurally equal to the type found in the dynamic value; it is allowed to be more specific than the type given. Thus, we can instantiate the type that is contained with the value in the dynamic value. For example, assume we require the result to be a function of the type `Int \rightarrow Int`:

\[
\text{idInt :: Maybe (Int \rightarrow Int)} \\
\text{idInt = case idDyn of} \\
\text{ (f :: Int \rightarrow Int) \rightarrow Just f} \\
\text{ - } \rightarrow \text{Nothing}
\]

Here, the expected type is unified with the type of the value from the dynamic value and when this succeeds, the value is implicitly coerced to the expected type and returned.
An interesting difference between the approaches in the two languages is that Haskell offers monomorphic dynamic typing via a library, while Clean offers a more expressive system with support for polymorphism via built-in language support. The advantage of the Clean approach is that the dynamic type system is defined on abstract syntax trees. These structures can be manipulated more freely in the implementation in the compiler, in contrast to a library which is restricted by the expressivity of the language itself. Also, a built-in system provides great flexibility in how dynamic typing is offered in the language syntactically. On the other hand, the Haskell approach does not need to extend the core language which reduces the complexity of the language and compiler.

In this chapter we investigate what it takes for a functional language to embed polymorphic dynamic typing as a library. We limit our scope to a system with predicative polymorphism. That is, bound variables can only be instantiated by base types without variables. Concretely, our contributions are the following:

- We show how to embed monomorphic dynamic typing in Haskell using generalised algebraic datatypes and discuss the difficulties in extending this approach to polymorphic dynamic typing (Section 6.2).
- We motivate the need for a dependently-typed language like Agda and define the embedding of polymorphic dynamic typing using structural equality on type representations (Section 6.3).
- We extend the Agda approach with an instance-of algorithm and give a complete proof of its correctness in Agda itself (Section 6.4).
- We define the corresponding cast function (Section 6.5).

Finally, we discuss related work (Section 6.6) and conclude with a brief discussion and future work (Section 6.7).

### 6.2 Embedding in a functional language

We first consider the embedding of monomorphic dynamic typing in Haskell using generalised algebraic datatypes (Section 6.2.1). Then, we discuss the difficulties in extending this approach to polymorphic dynamic typing (Section 6.2.2).

#### 6.2.1 Monomorphic dynamic typing

As alluded to in the introduction, a dynamic value constitutes a value and a representation of its type. Hence, for us to describe a dynamic value, we first need a datatype that describes types. A naive approach is to define a universe for the representation of types as a vanilla datatype:

```haskell
data U = INT
    | PAIR U U
    | U =>: U
```

The universe describes integer, pair, and function types respectively. Then, a dynamic value is defined as follows:
6 Embedding polymorphic dynamic typing

\[ \text{data } \textit{Dyn} = \forall \alpha . \text{Dyn } U \alpha \]

A dynamic value is a black box, hence, the type of the value contained is existentially quantified\(^1\). The main problem with this approach already becomes apparent. How is it captured that the value of type \(U\) represents the type \(\alpha\) in the definition of \(\textit{Dyn}\)? This becomes even more clear when we write down the type of the function that casts a dynamic value to a specified type:

\[ \text{cast } :: U \rightarrow \text{Dyn} \rightarrow \text{Maybe } \alpha \]

Again, the relation between the resulting value of this cast function and the required type is missing.

In Haskell, generalised algebraic datatypes (Peyton Jones et al., 2006) provide a solution. We define \(U\) again, but now include a type parameter that describes the type that the universe represents:

\[ \text{data } U :: \star \rightarrow \star \text{ where } \]

\[ \text{INT } :: U \text{ Int} \]

\[ \text{PAIR } :: U \alpha \rightarrow U \beta \rightarrow U (\alpha, \beta) \]

\[ (\Rightarrow:) :: U \alpha \rightarrow U \beta \rightarrow U (\alpha \rightarrow \beta) \]

A value of type \(U \alpha\) describes the type \(\alpha\) that it represents. Hence, when we define \(\textit{Dyn}\) again, the type of the value is visible in the representation that is contained in the dynamic value:

\[ \text{data } \textit{Dyn} = \forall \alpha . \text{Dyn } (U \alpha) \alpha \]

In order to unwrap the existentially quantified value from the dynamic value, we need to compare the contained representation with a representation and then prove that these describe the same type. This proof of equality is defined by a generalised algebraic datatype, stating that both type parameters are equal:

\[ \text{data } (\Rightarrow:) :: \star \rightarrow \star \rightarrow \star \text{ where } \]

\[ \text{Refl } :: \alpha \Rightarrow: \alpha \]

The function that decides if two representations describe equal types uses structural recursion on both arguments:

\[
\begin{align*}
\text{decU } :: & \ U \alpha \rightarrow U \beta \rightarrow \text{Maybe } (\alpha \Rightarrow: \beta) \\
\text{decU } \text{INT} & \text{ INT} = \text{Just Refl} \\
\text{decU } (\text{PAIR } u1 \ u1') (\text{PAIR } u2 \ u2') & = \text{do} \ \text{Refl} \leftarrow \text{decU } u1 \ u2 \\
& \ \text{Refl} \leftarrow \text{decU } u1' \ u2' \\
& \ \text{Just Refl} \\
\text{decU } (u1 \Rightarrow: u1') (u2 \Rightarrow: u2') & = \text{do} \ \text{Refl} \leftarrow \text{decU } u1 \ u2 \\
& \ \text{Refl} \leftarrow \text{decU } u1' \ u2' \\
& \ \text{Just Refl} \\
\text{decU } _ & \ _ = \text{Nothing}
\end{align*}
\]

\(^1\)Ironically, this is denoted in Haskell using universal quantification inside a datatype.
Note that we need to explicitly pattern match on the \textit{Refl} constructor obtained from recursion to actually deploy the proof. We use \texttt{do}-notation to conveniently combine these results.

Using this function, we are able to define the \texttt{cast} function which performs the unwrapping:

\[
\begin{align*}
\text{cast} &\colon U \alpha \to \text{Dyn} \to \text{Maybe} \alpha \\
\text{cast } u1 \ (\text{Dyn } u2 \ x) &= \texttt{do} \ 	ext{Refl} \leftarrow \text{decU } u1 \ u2 \\
&\quad \texttt{Just } x
\end{align*}
\]

When the required representation describes the same type as the representation in the dynamic value, the proof \textit{Refl} tells us that we can safely return the value that is contained in the dynamic.

### 6.2.2 Difficulties

Generalised algebraic datatypes allow us to attach an actual type to a representation of a monomorphic type. But can we extend this approach to a representation of polymorphic types? This requires a way to bind and reference variables. Typically, this is achieved either by using De Bruijn indices (De Bruijn, 1972) or through higher-order abstract syntax (Pfenning and Elliot, 1988).

Since we need occurrences of the same variable to describe the same type, either approach requires a type representation to carry additional administration in the form of an environment. A dynamic value demands this environment to be closed since a value cannot be related to a representation in the presence of free variables. Unfortunately, this prevents us from comparing such representations. Since environments are closed, equal references from different representations are not known to describe the same type since they involve different environments.

The only way to circumvent these troubles is to postpone the use of an environment and hence the attachment of types to representations. This asks for a separation between a universe for the representation of types and its interpretation as a type. The intuition is that the separation allows us to first perform the desired operations on representations, after which we perform the interpretation at the latest moment. This allows us, for instance, to compare representations without having any attached interpretation in the way.

Ideally, we would like to define a function that interprets a representation and returns the type that it describes. Haskell provides some way to do type-level computations via generalised algebraic datatypes and type families (Schrijvers et al., 2008). While there are some possibilities to embed polymorphic dynamic typing in Haskell by making heavy use of these tools, we believe that a dependently-typed language provides a more natural approach.

### 6.3 Embedding in a dependently-typed language

In this section we use Agda (Norell, 2007) and discuss how monomorphic dynamic typing can be embedded in this language (Section 6.3.1). Then, we show how to elegantly embed polymorphic dynamic typing, for now limiting ourselves to
using structural equality of representations (Section 6.3.2). We will return to an embedding of polymorphic dynamic typing using an instance-of algorithm later in Section 6.4.

### 6.3.1 Monomorphic dynamic typing

We begin by defining a universe to represent monomorphic types:

```agda
data U : Set where
  NAT : U
  PAIR : U → U → U
  _⇒_ : U → U → U
```

The difference with the representation from Section 6.2.1 is that the interpretation of such a representation (i.e., the type that a representation describes) is detached. Agda, being a dependently-typed language, allows us to obtain the corresponding type by a function `el` that computes the elements of the universe `U`:

```agda
el : U → Set
el NAT         = Nat
el (PAIR u u') = Pair (el u) (el u')
el (u ⇒ u')    = el u → el u'
```

This function returns a type when given a value. The base case returns the monomorphic type `Nat` whereas the other branches for pairs and functions recurse while constructing a pair or function type. Then, a dynamic value constitutes a representation with a value of the interpreted type:

```agda
data Dyn : Set where
dyn : (u : U) → el u → Dyn
```

A cast function needs a proof that a provided representation is equal to the representation that is contained in the dynamic value, before unwrapping its value:

```agda
data _≡_ {a : Set} (x : a) : a → Set where
  Refl : x ≡ x
```

This datatype is similar to the Haskell datatype `≡` from Section 6.2.1, although there is one important difference: this equality states that two values are equal while the Haskell datatype states that two types are equal. Here, we only need a proof on the value level since a representation `U` does not directly describe the type it represents. The function that decides equality of representations is defined as follows:

```agda
decU : (u1 u2 : U) → Maybe (u1 ≡ u2)
decU NAT       NAT = Just Refl
decU (PAIR u1 u1') (PAIR u2 u2') with decU u1 u2 | decU u1' u2'
decU (PAIR u1 u1') (PAIR .u1 .u1') | Just Refl | Just Refl = Just Refl
...  | _ | _ = Nothing
```
6.3 Embedding in a dependently-typed language

\[
\text{decU} \ (u1 \Rightarrow u1') \ (u2 \Rightarrow u2') \ \text{with} \ \text{decU} \ u1 \ u2 \ | \ \text{decU} \ u1' \ u2' \\
\quad \ldots \quad \text{Just Refl} \quad \text{Just Refl} = \text{Just Refl} \\
\text{decU} \_ \_ = \text{Nothing}
\]

This function takes two representations as arguments, described in its type by \((u1 \ u2 : U)\) as shorthand for \((u1 : U) \to (u2 : U)\), and returns a possible proof of their equality. Coming up with a proof in the case of integer representations is easy. In the branches for pairs and functions we pair-wise recurse using the keyword \text{with} and pattern match on the results. Note that when pattern matching on a proof \text{Refl}, we have to restate the branch and use the dot-notation to explicitly state that we have learned that the two compared elements are equal. Otherwise, we use shorthand notation \ldots to restate the original branch before we come to pattern matching the other possible results of recursion.

Next, we define the cast function:

\[
\text{cast} : (u1 : U) \to \text{Dyn} \to \text{Maybe} \ (\text{el} \ u1) \\
\text{cast} \ u1 \ (\text{dyn} \ u2 \ \text{x}) \ \text{with} \ \text{decU} \ u1 \ u2 \\
\quad \text{cast} \ u1 \ (\text{dyn} \ .u1 \ \text{x}) \ | \ \text{Just Refl} = \text{Just x} \\
\quad \ldots \quad \_ \_ = \text{Nothing}
\]

When the function \text{decU} gives the proof \text{Refl} that both representations are equal, we can return the value that is contained in the dynamic value. Note that although the proof only states the equality of the representation values, we also learn that \text{el} \ u1 equals \text{el} \ u2.

### 6.3.2 Polymorphic dynamic typing

Next, we extend the previous approach to polymorphic dynamic typing. First, we need a constructor for variables in our universe \(U\):

\[
data \ U \ (n : \text{Nat}) : \text{Set} \ where \\
\text{NAT} : U n \\
\text{PAIR} : U n \to U n \to U n \\
\_ \Rightarrow \_ : U n \to U n \to U n \\
\text{VAR} : \text{Fin} n \to U n
\]

The universe \(U\) now also contains a parameter that indicates the number of variables that a representation can use at most. The datatype \(\text{Fin} \ n\) describes \(n\) possible variable references:

\[
data \ \text{Fin} : \text{Nat} \to \text{Set} \ where \\
\text{Fz} : \text{forall} \ \{n : \text{Nat}\} \to \text{Fin} \ (\text{Succ} \ n) \\
\text{Fs} : \text{forall} \ \{n : \text{Nat}\} \to \text{Fin} \ n \to \text{Fin} \ (\text{Succ} \ n)
\]

To be able to relate values to representations, we introduce a new universe \(V\) that closes a representation that contains variables:

\[
data \ V : \text{Set} \ where \\
\text{FORALL} : \text{forall} \ \{n : \text{Nat}\} \to U n \to V
\]
Using the described universes $U$ and $V$, the type of the polymorphic identity function can be represented as $\text{FORALL} (\text{VAR } Fz \Rightarrow \text{VAR } Fz)$. In fact, there are infinitely many representations of this type; we can use any $\text{Fin}$ value as a variable reference as long as both references are the same. Note that the type of a representation describes the number of variables it can use at most, which is not necessarily the same as the exact number of variables it actually uses. For instance, the above representation $\text{VAR } Fz \Rightarrow \text{VAR } Fz$ can be given the type $U (\text{Succ } (\text{Succ } \text{Zero}))$, but also $U (\text{Succ } (\text{Succ } \text{Zero}))$, and so on.

Since a representation no longer directly describes its corresponding type, as in Section 6.3.1, we use an interpretation function to compute the desired type for the value in the dynamic value:

$$eLV : V \rightarrow \text{Set}$$
$$eLV (\text{FORALL} \{n\} u) = \text{forall } \{\text{env} : \text{Env } n\} \rightarrow eLU u \text{ env}$$

The interpretation introduces a quantifier to bind all variables occurring in the representation using an environment as an implicit argument. The environment is modelled as follows:

```haskell
data Env : Nat \rightarrow \text{Set} where
  Nil : Env Zero
  Cons : forall \{n : Nat\} \rightarrow U Zero \rightarrow \text{Env } n \rightarrow \text{Env } (\text{Succ } n)
```

The environment contains representations that do not use any variables, this enforces the fact that we are dealing with a system with predicative polymorphism.

Finding an entry in an environment using a variable reference is straightforward:

$$\text{findInEnv} : \text{forall } \{n : \text{Nat}\} \rightarrow \text{Fin } n \rightarrow \text{Env } n \rightarrow U \text{ Zero}$$
$$\text{findInEnv } Fz (\text{Cons } u) = u$$
$$\text{findInEnv } (\text{Fs } i) (\text{Cons } _\text{env}) = \text{findInEnv } i \text{ env}$$

In the base case for $Fz$ we take the head entry, and otherwise we recurse with the tail of the environment. Note that we do not need to provide a branch for the empty environment since the constructors of the $\text{Fin}$ type do not permit a reference to an empty environment.

The interpretation function of the universe $U$ takes such an environment as an additional argument when computing its elements:

$$eLU : \text{forall } \{n : \text{Nat}\} \rightarrow U n \rightarrow \text{Env } n \rightarrow \text{Set}$$
$$eLU \text{ NAT } _\text{ } = \text{Nat}$$
$$eLU (\text{PAIR } u u') \text{ env } = \text{Pair } (eLU u \text{ env}) (eLU u' \text{ env})$$
$$eLU (u \Rightarrow u') \text{ env } = eLU u \text{ env } \rightarrow eLU u' \text{ env}$$
$$eLU (\text{VAR } i) \text{ env } = eLU0 (\text{findInEnv } i \text{ env})$$

The representations of integers, pairs, and functions map to their respective types. In the case of a variable we use the environment that is passed along to obtain the type that this variable refers to. Because we quantify over representations, we have to interpret its result again. We stratify the interpretation functions on
6.3 Embedding in a dependently-typed language

the universe U to ensure termination, hence, we define elU0 separately to interpret representations without variables:

\[
\begin{align*}
\text{elU0} & : U \text{ Zero} \to \text{Set} \\
\text{elU0} \text{ NAT} & = \text{Nat} \\
\text{elU0} \text{ (PAIR } u u') & = \text{Pair (elU0 } u \text{) (elU0 } u') \\
\text{elU0} \text{ (} u \Rightarrow u' \text{)} & = \text{elU0 } u \to \text{elU0 } u' \\
\text{elU0} \text{ (VAR } ()) & \\
\end{align*}
\]

The function elU0 recurses over the structure of the argument representation and produces types along the way, similar to elU. Since we know that such a representation cannot contain any variables (i.e., the type Fin Zero is uninhabited), we define the final branch for VAR as an impossible pattern using ()

Then, using our new universes and corresponding interpretation functions, we redefine the datatype Dyn as follows:

\[
\text{data Dyn} : \text{Set where} \\
\text{dyn} : (v : V) \to \text{elV } v \to \text{Dyn}
\]

Again, in order to unwrap the value that is contained in a dynamic value later in the cast function, we continue by defining decidable equality on our representations V and U. First, we consider the universe V:

\[
\begin{align*}
\text{decV} & : (v1 v2 : V) \to \text{Maybe (v1} \equiv \text{v2)} \\
\text{decV} \text{ (FORALL } {n} \_ \text{) (FORALL } {m} \_ \text{) with cmp } n \text{ m} \\
\text{decV} \text{ (FORALL } {n} \text{ u1) (FORALL } {n} \text{ u2) | EQ with decU } u1 \text{ u2} \\
\text{decV} \text{ (FORALL } .u1 \text{) (FORALL u1) | EQ | Just Refl = Just Refl} \\
... & | _ = \text{Nothing} \\
\text{decV} \text{ (FORALL } \_ \text{) (FORALL } \_ \text{) | _ = Nothing}
\end{align*}
\]

We consider representations to be equal only if they can use the same number of variables and their contained base universe is equal as well. The former is verified using a helper function named cmp and a datatype Cmp that describes the difference between the two values:

\[
\begin{align*}
\text{cmp} & : (n m : \text{Nat}) \to \text{Cmp } n m \\
\text{cmp Zero} & \text{ Zero } = \text{EQ} \\
\text{cmp Zero} & \text{ (Succ } m\text{) } = \text{LT (Succ } m\text{)} \\
\text{cmp (Succ } n\text{)} & \text{ Zero } = \text{GT (Succ } n\text{)} \\
\text{cmp (Succ } n\text{)} & \text{ (Succ } m\text{) with cmp } n \text{ m} \\
\text{cmp (Succ } n\text{)} & \text{ (Succ } .n\text{) } | \text{EQ } = \text{EQ} \\
\text{cmp (Succ } n\text{)} & \text{ (Succ } .(n + k)\text{) } | \text{LT } k \text{ } = \text{LT } k \\
\text{cmp (Succ } .(m + k)\text{)} & \text{ (Succ } m\text{) } | \text{GT } k \text{ } = \text{GT } k \\
\text{data Cmp} & : \text{Nat } \to \text{Nat } \to \text{Set where} \\
\text{EQ} & : \text{forall } \{n : \text{Nat}\} \to \text{Cmp } n n \\
\text{LT} & : \text{forall } \{n : \text{Nat}\} \to (k : \text{Nat}) \to \text{Cmp } n \text{ (n + k)} \\
\text{GT} & : \text{forall } \{n : \text{Nat}\} \to (k : \text{Nat}) \to \text{Cmp } (n + k) \text{ n}
\end{align*}
\]

The function cmp is straightforwardly defined by induction on both its arguments.
Then, we define decidable equality on the base universe $U$:

$$\text{decU} : \forall \{n : \text{Nat}\} \to (u1 \ u2 : U n) \to \text{Maybe} \ (u1 \equiv u2)$$

$$\text{decU} \ \text{NAT} \ \text{NAT} = \text{Just Refl}$$

$$\text{decU} \ (\text{PAIR} \ u1 \ u1') \ (\text{PAIR} \ u2 \ u2') \ \text{with} \ \text{decU} \ u1 \ u2 \ | \ \text{decU} \ u1' \ u2'$$

$$\text{decU} \ (\text{PAIR} \ u1 \ u1') \ (\text{PAIR} \ .u1 \ .u1') \ | \ \text{Just Refl} \ | \ \text{Just Refl} = \text{Just Refl}$$

$$\vdots \ | \ _ \ | \ _ = \text{Nothing}$$

$$\text{decU} \ (u1 \Rightarrow u1') \ (u2 \Rightarrow u2') \ \text{with} \ \text{decU} \ u1 \ u2 \ | \ \text{decU} \ u1' \ u2'$$

$$\text{decU} \ (u1 \Rightarrow u1') \ (.u1 \Rightarrow .u1') \ | \ \text{Just Refl} \ | \ \text{Just Refl} = \text{Just Refl}$$

$$\vdots \ | \ _ \ | \ _ = \text{Nothing}$$

$$\text{decU} \ (\text{VAR} \ i) \ (\text{VAR} \ j) \ \text{with} \ \text{decFin} \ i \ j$$

$$\text{decU} \ (\text{VAR} \ i) \ (\text{VAR} \ .i) \ | \ \text{Just Refl} = \text{Just Refl}$$

$$\vdots \ | \ _ = \text{Nothing}$$

$$\text{decU} \ _ \ _ = \text{Nothing}$$

This definition greatly resembles the definition of $\text{decU}$ in Section 6.3.1, but we add the branch for variables where we use the function $\text{decFin}$:

$$\text{decFin} : \forall \{n : \text{Nat}\} \to (i : \text{Fin} n) \to \text{Maybe} \ (i \equiv j)$$

$$\text{decFin} \ \text{Fz} \ \text{Fz} = \text{Just Refl}$$

$$\text{decFin} \ (\text{Fs} \ i) \ (\text{Fs} \ j) \ \text{with} \ \text{decFin} \ i \ j$$

$$\text{decFin} \ (\text{Fs} \ i) \ (\text{Fs} \ .i) \ | \ \text{Just Refl} = \text{Just Refl}$$

$$\vdots \ | \ _ = \text{Nothing}$$

$$\text{decFin} \ _ \ _ = \text{Nothing}$$

Then, we are finally able to define the cast function on the universe that represents polymorphic types:

$$\text{cast} : (v1 : V) \to \text{Dyn} \to \text{Maybe} \ (\text{elV} \ v1)$$

$$\text{cast} \ v1 \ (\text{dyn} \ v2 \ x) \ \text{with} \ \text{decV} \ v1 \ v2$$

$$\text{cast} \ v1 \ (\text{dyn} \ .v1 \ x) \ | \ \text{Just Refl} = \text{Just x}$$

$$\vdots \ | \ _ = \text{Nothing}$$

This definition is much like the definition of $\text{cast}$ in Section 6.3.1: we compare the given representation with the representation found in the dynamic value and return the value if the proof tells us that the representations are structurally equal.

### 6.4 Instance-of algorithm

In the previous section we showed how to embed polymorphic dynamic typing in Agda. However, we used structural equality on representations to define the cast function. Typically, we would want to unify the required representation with the one found in the dynamic value. However, caution is advised since we can only unwrap a value when the required representation is an instance of (i.e., does not describe a more general type than) the representation at hand.

Recalling the example from the introduction, consider the polymorphic identity function wrapped in a dynamic value using our universe for the representation of polymorphic types:
idType : V
idType = FORALL \{ Succ Zero \} (VAR Fz ⇒ VAR Fz)

idDyn : Dyn
idDyn = dyn idType (λ x → x)

The representation idType must explicitly provide the implicit argument of the FORALL constructor since it is otherwise unclear to Agda what the number of variables is that the representation can use. Here, we choose the exact number of variables that the representation uses, Succ Zero. Recall that we could have also chosen any greater number, just like we discussed earlier in Section 6.3.2. Additionally, we define a representation of the type of the increment function on integers:

incType : V
incType = FORALL \{ Zero \} (NAT ⇒ NAT)

Then, the following expression must yield an identity function on integers since the required representation is an instance of the one found in the dynamic value:

cast incType idDyn

The other way around, consider a dynamic value that contains the increment function on integers:

incDyn : Dyn
incDyn = dyn incType (λ x → x + 1)

When we try to unwrap the function from this dynamic value, we are not allowed to cast it to the more general type of the polymorphic identity function. For example, the following expression must fail and return Nothing:

cast idType incDyn

At first sight it seems like we can take one of the well-known algorithms for unification as a starting point; this will give us a substitution that when applied to the two representations results in the same representation. For the instance-of algorithm we need a substitution that applied to the one results in the other, so the substitution obtained from unification should suffice. However, the situation is a little more subtle: unification tries to unify two representations into a common representation, whereas the instance-of algorithm must bring us from the one representation to the other. This tells us the following about the required substitution: its domain constitutes only those variables that occur in the one representation, and its codomain only variables from the other. Consequently, we do not need to recursively apply substitutions nor to perform an occurs-check. We only need to enforce that equal variables map to equal types while constructing a substitution. In fact, the instance-of algorithm closely resembles the algorithm defined for matching rewrite rules to terms as described in Chapter 3.

In this section we first define substitutions and their application (Section 6.4.1). Then, we define an accumulating instance-of function that gives us a substitution.
and prove the correctness of the algorithm in Agda itself (Section 6.4.2). For now, we are only concerned with representations in the base universe $U$. Later in Section 6.5, we will see how the cast function deals with representations in the universe $V$.

### 6.4.1 Substitution

The core of the instance-of algorithm lies in substitution. This is modelled as an associative list that possibly maps variables to representations that may contain other variables:

```agda
data Subst (m : Nat) : Nat → Set where
  Nil : Subst m Zero
  Cons : forall {n : Nat} →
    Maybe (U m) → Subst m n → Subst m (Succ n)
```

Note that we can use the same constructor names as in the datatype `Env` from Section 6.3.2 since Agda supports constructor overloading. The datatype `Subst` is parameterised by $m$, the number of variables that the stored representations can use, and indexed by $n$, the length of the substitution.

The empty substitution is easily constructed by performing induction on the length of the substitution and inserting `Nothing` values at every position:

```agda
empty : forall {n m : Nat} → Subst m n
empty {Zero}  = Nil
empty {Succ n} = Cons Nothing empty
```

A substitution is applied to a representation using the following function:

```agda
apply : forall {n : Nat} → Subst n n → U n → U n
apply _ NAT = NAT
apply subst (PAIR u u’) = PAIR (apply subst u) (apply subst u’)
apply subst (u ⇒ u’) = apply subst u ⇒ apply subst u’
apply subst (VAR i) with findInSubst i subst
  ... | Just u = u
  ... | _ = VAR i
```

The function `apply` recurses over the structure of the representation until it encounters a variable. There, it uses a function that finds a value in a substitution. If there is no such value present, the variable is left untouched. Hence, the substitution that is provided to `apply` couples the type of its values to its length: it contains values of type $U n$ and is of length $n$. The function `findInSubst` is similar to `findInEnv` as defined in Section 6.3.2:

```agda
findInSubst : forall {n m : Nat} → Fin n → Subst m n → Maybe (U m)
findInSubst Fz   (Cons mu _)  = mu
findInSubst (Fs i) (Cons _ subst) = findInSubst i subst
```

Here, the length of the substitution must be decoupled from the type of the values in the substitution to be able to perform induction on the length of the substitution without having to change the type of its values in recursion.
6.4 Instance-of algorithm

6.4.2 Accumulating instance-of function

Now that we have defined the type of substitutions and their application, we continue by defining the actual instance-of function. It determines if the first argument is an instance of the second argument and returns the witnessing substitution. Consequently, applying the resulting substitution to the second argument must give us the first argument back. It uses an accumulating parameter for the substitution constructed thusfar:

\[
iofAcc : \forall \{ n : \text{Nat} \} \rightarrow \n \rightarrow \n \rightarrow \text{Subst} n n \rightarrow \text{Maybe} (\text{Subst} n n)
\]

\[
iofAcc \text{ NAT } \text{ NAT } \text{ subst } = \text{ Just subst}
\]

\[
iofAcc (\text{PAIR u1 u1'}) (\text{PAIR u2 u2'}) \text{ subst with } \text{iofAcc} u1 u2 \text{ subst}
\]

\[
\ldots \mid \text{ Just subst' } = \text{iofAcc} u1' u2' \text{ subst'}
\]

\[
\ldots \mid _ - = \text{Nothing}
\]

\[
iofAcc (u1 \Rightarrow u1') (u2 \Rightarrow u2') \text{ subst with } \text{iofAcc} u1 u2 \text{ subst}
\]

\[
\ldots \mid \text{ Just subst' } = \text{iofAcc} u1' u2' \text{ subst'}
\]

\[
\ldots \mid _ - = \text{Nothing}
\]

\[
iofAcc u1 (\text{VAR i}) \text{ subst } = \text{ update i u1 subst}
\]

\[
iofAcc _ - - = \text{Nothing}
\]

Note that although the two arguments are required to use the same number of variables, in practice this is not always the case. We have seen an example of this earlier when we described that incType is an instance of idType; the former uses Zero variables whereas the latter uses Succ Zero variables. In Section 6.5 we take care of this issue. For now we assume the two arguments of iofAcc to use the same number of variables. Also, observe that the type of the resulting substitution relates the type of its values with its length, just like the function apply from Section 6.4.1 desires. The function iofAcc defines that an integer representation is an instance of an integer representation, and we return the accumulated substitution. A pair is an instance of another pair, but only if both their elements are pair-wise instances of each other. Since we use an accumulating parameter, we thread the substitution. The case for functions proceeds in the same fashion. Finally, every representation is an instance of a variable, in that case we update the substitution, in all other cases we return Nothing. The update function makes sure that equal variables map to equal types, thus keeping the substitution consistent:

\[
\text{update} : \forall \{ n m : \text{Nat} \} \rightarrow \n \rightarrow \text{Subst} m n \rightarrow \text{Maybe} (\text{Subst} m n)
\]

\[
\text{update Fz} u (\text{Cons Nothing subst}) = \text{ Just (Cons (Just u) subst)}
\]

\[
\text{update Fz} u (\text{Cons (Just u') _ with } \text{decU'} u u')
\]

\[
\text{update Fz} u (\text{Cons (Just .u) subst}) \mid \text{ Inl Refl } = \text{ Just (Cons (Just u) subst)}
\]

\[
\ldots \mid \text{ Inr _ } = \text{Nothing}
\]

\[
\text{update (Fs i) u (Cons mu' subst) with } \text{update i u subst}
\]

\[
\ldots \mid \text{ Just subst' } = \text{ Just (Cons mu' subst')}
\]

\[
\ldots \mid _ - = \text{Nothing}
\]

This function proceeds by induction on the variable reference and the substitution. If we are considering the top reference and there is not yet a value for that variable
reference (i.e., the substitution contains a \texttt{Nothing} value) we simply insert it. If there is a value present in the substitution, we only succeed if the encountered value is the same representation as the provided value. Otherwise, the substitution is not consistent and we fail by returning \texttt{Nothing}. In the final case we recurse with the tail of the substitution to update the value. The function \texttt{decU'} is similar to the function \texttt{decU} from Section 6.3.2:

\[
\texttt{decU'} : \forall \{ n : \text{Nat} \} \rightarrow (u1 u2 : U n) \rightarrow \text{Either} (u1 \equiv u2) (u1 \neq u2)
\]

The function \texttt{decU'} follows the same recipe as \texttt{decU}: it performs induction on both \texttt{u1} and \texttt{u2}. The difference is that instead of returning \texttt{Nothing} in the off-diagonal cases, this variant also provides a proof of inequality. Such a proof is modelled as a function that given a proof of equality, results in a value of the \texttt{Empty} type which has no constructors:

\[
\neq \neq : \forall \{ a : \text{Set} \} \rightarrow a \rightarrow a \rightarrow \text{Set}
\]

\[
x \neq y = x \equiv y \rightarrow \text{Empty}
\]

\[
data \text{Empty} : \text{Set}
\]

\[
\text{where}
\]


We use \texttt{decU'} instead of \texttt{decU} in the definition of \texttt{update} since we need such a proof of inequality to prove the correctness of the instance-of algorithm. The complete definition of this function can be found in Appendix 6.B.1.

Then, all that is left to do is to start the accumulating instance-of function:

\[
\texttt{iof} : \forall \{ n : \text{Nat} \} \rightarrow U n \rightarrow U n \rightarrow \text{Maybe} (\text{Subst n n})
\]

\[
\texttt{iof u1 u2} = \texttt{iofAcc u1 u2 empty}
\]

We provide the empty substitution to \texttt{iofAcc}, which concludes the definition of the instance-of algorithm.

To be able to define the cast function later in Section 6.5, we need to prove that the instance-of algorithm is correct: applying a substitution to the one representation results in the other. We will use this proof in the cast function to coerce the value contained in the dynamic value to the desired type. The proof of correctness is described in terms of the accumulating instance-of function:

\[
\texttt{iofCorrect} : \forall \{ n : \text{Nat} \} \rightarrow (u1 u2 : U n) \rightarrow (\text{subst subst'} : \text{Subst n n}) \rightarrow \text{iofAcc u1 u2 subst} \equiv \text{Just subst'} \rightarrow \text{apply subst'} u2 \equiv u1
\]

The proof proceeds by induction on \texttt{u1} and \texttt{u2}. The cases where \texttt{iofAcc} produces \texttt{Nothing} are easily closed using the argument proof that states that it does succeed. The cases for integers are straightforward whereas the cases for pairs and functions present us with more work. There, we have to provide proof that consistency of the substitution is maintained by the update function while threading it in the instance-of algorithm. Similarly, in the case of a variable, we have to provide proof that no information is lost in the substitution and that any value that is updated can always be retrieved. The complete definition of the proof can be found in Appendix 6.B.2.
For convenience, we define a function that uses iof and wraps its result with the proof of correctness:

\[
\text{iofCorrectness} : \forall \{n : \text{Nat}\} \rightarrow (u1 \ u2 : \text{U } n) \rightarrow \text{Maybe} (\exists \text{Subst } n \ n) \rightarrow (\lambda \text{subst } \rightarrow \text{apply subst } u2 \equiv u1))
\]

\[
\text{iofCorrectness } u1 \ u2 \text{ with inspect } (\text{iof } u1 \ u2)
\]

... | Just subst with-≡ eq =

... | _ with-≡ _ = Nothing

The possible result of this function consists of a substitution and a corresponding proof. The relation between the two is modelled using a special kind of pair where the type of the second value may depend on the first value, named a dependent pair:

\[
\text{data} \text{ Exists } (a : \text{Set}) \ (f : a \rightarrow \text{Set}) : \text{Set where}
\]

\[
\text{Witness} : (x : a) \rightarrow f x \rightarrow \text{Exists } a \ f
\]

When iof succeeds and results in a substitution, we need to remember this such that we can provide proof of this fact as an argument to iofCorrect. To this end, we define a nifty function named inspect:

\[
\text{inspect} : \forall \{a : \text{Set}\} \rightarrow (x : a) \rightarrow \text{Inspect } x
\]

\[
\text{inspect } x = x \text{ with-≡ Refl}
\]

\[
\text{data} \text{ Inspect } \{a : \text{Set}\} \ (x : a) : \text{Set where}
\]

\[
\_ \text{with-≡} _ : (y : a) \rightarrow x \equiv y \rightarrow \text{Inspect } x
\]

This function inspects the provided value and wraps it with an equality proof.

### 6.5 Cast

In Section 6.3, we defined a straightforward cast function in Agda that uses structural equality on representations. In the previous section we have stated our desire to verify if the given representation is an instance of the representation found in the dynamic value. If this is the case, we know that it is safe to unwrap the corresponding value from a dynamic value.

In this section we show how to define such a cast function. First, let us recall its type:

\[
\text{cast} : (v1 : V) \rightarrow \text{Dyn} \rightarrow \text{Maybe } (elV v1)
\]

The representation that is found in the dynamic value, say v2 containing base universe u2, can possibly use a different number of variables than the given representation v1 containing base universe u1. Since the instance-of algorithm expects representations that use the same number of variables, we first need to align the representations (Section 6.5.1). Then, in order to unwrap the value from the dynamic value, we have to transform that value from type elV v2 to elV v1. The
trick is to use the form of the result of their interpretation using \( \text{elV} \), as defined in Section 6.3.2, to our advantage. The interpretation function introduces a quantifier that binds all variables using an environment. Hence, this environment is our point of entry to instantiate variables in \( u_2 \) using a substitution \( \text{subst} \) resulting from the instance-of algorithm (Section 6.5.2). This will give us a value with a type that is described by the representation \( \text{apply subst} \ u_2 \), which suits the proof of correctness of the instance-of algorithm from Section 6.4.2 and allows us to coerce a value to a type described by \( u_1 \). Finally, we put all of this together in a single cast function (Section 6.5.3).

### 6.5.1 Alignment

Two representations are aligned by weakening the one representation that can use fewer variables. The following function increments the number of variables by one:

\[
\text{weaken : forall } \{ n : \text{Nat} \} \rightarrow U \ n \rightarrow U \ (\text{Succ}\ n)
\]

\[
\text{weaken \ NAT} = \text{NAT}
\]

\[
\text{weaken \ (PAIR} \ u \ u') = \text{PAIR} \ (\text{weaken } u) \ (\text{weaken } u')
\]

\[
\text{weaken \ (u } \Rightarrow u') = \text{weaken } u \Rightarrow \text{weaken } u'
\]

\[
\text{weaken \ (VAR} i) = \text{VAR} \ (\text{inj } i)
\]

This is a function that only changes the type of a representation, not its value. It recurses on the structure of the representation and in the case of a variable the reference itself is weakened:

\[
\text{inj : forall } \{ n : \text{Nat} \} \rightarrow \text{Fin} \ n \rightarrow \text{Fin} \ (\text{Succ}\ n)
\]

\[
\text{inj \ Fz} = \text{Fz}
\]

\[
\text{inj \ (Fs} i) = \text{Fs} \ (\text{inj } i)
\]

The type of \( \text{Fz} \) allows us to choose any \( n \), in this case we choose \( \text{Succ}\ n \).

We can also iteratively weaken a representation an arbitrary number of times:

\[
\text{weakenByK : forall } \{ n : \text{Nat} \} \rightarrow (k : \text{Nat}) \rightarrow U \ n \rightarrow U \ (n + k)
\]

\[
\text{weakenByK } \{ n \} \text{ Zero } u \text{ rewrite plusZero } n = u
\]

\[
\text{weakenByK } \{ n \} \ (\text{Succ} k) u \text{ rewrite plusSucc } n k = \text{weakenByK} k \ (\text{weaken } u)
\]

Here, we nest \( k \) applications of the \text{weaken} function. Additionally, we need two proofs to mold the types in the right shape:

\[
\text{plusZero : (n : \text{Nat})} \rightarrow n + \text{Zero} \equiv n
\]

\[
\text{plusSucc : (n m : \text{Nat})} \rightarrow n + \text{Succ} m \equiv \text{Succ} (n + m)
\]

These functions are defined in Appendix 6.C.1 and perform straightforward induction on their arguments. They are deployed by using the keyword \text{rewrite}, which rewrites the types occurring in the current context.

Now that we know how to weaken representations, we will use this to define two dual functions that coerce values that are related to such weakened representations. We name these functions \text{shrink} and \text{grow}; they coerce values from \( \text{elU} \ (\text{weakenByK} k \ u) \ \text{env} \) to \( \text{elU} \ u \ \text{env} \) and vice versa. Obviously, the environments
used with either representations cannot be the same since the representations can use a different number of variables. Hence, to perform the coercion in both directions, we have to be able to make environments larger and smaller. Then, we provide a larger environment where a smaller one is desired, and vice versa. Consider the following two functions:

\[
\text{snoc} : \text{forall } \{n : \text{Nat}\} \rightarrow \text{Env } n \rightarrow \text{Env } (\text{Succ } n)
\]

\[
\text{snoc } \text{Nil} = \text{Cons } \text{NAT} \text{ Nil}
\]

\[
\text{snoc } (\text{Cons } u \text{ env}) = \text{Cons } u \text{ (snoc } \text{ env})
\]

\[
\text{init} : \text{forall } \{n : \text{Nat}\} \rightarrow \text{Env } (\text{Succ } n) \rightarrow \text{Env } n
\]

\[
\text{init } (\text{Cons } _{-} \text{ Nil}) = \text{Nil}
\]

\[
\text{init } (\text{Cons } u \text{ (Cons } u' \text{ env})) = \text{Cons } u \text{ (init } (\text{Cons } u' \text{ env}))
\]

The function \(\text{snoc}\) adds a stand-in value \(\text{NAT}^2\). Of course, we can also do this iteratively an arbitrary number of times:

\[
\text{snocByK} : \text{forall } \{n : \text{Nat}\} \rightarrow (k : \text{Nat}) \rightarrow \text{Env } n \rightarrow \text{Env } (n + k)
\]

\[
\text{snocByK} \{n\} \text{ Zero } \text{ env rewrite } \text{plusZero } n = \text{env}
\]

\[
\text{snocByK} \{n\} \text{ (Succ } k \text{) env rewrite } \text{plusSucc } n \text{ k} = \text{snocByK } k \text{ (snoc } \text{ env})
\]

\[
\text{initByK} : \text{forall } \{n : \text{Nat}\} \rightarrow (k : \text{Nat}) \rightarrow \text{Env } (n + k) \rightarrow \text{Env } n
\]

\[
\text{initByK} \{n\} \text{ Zero } \text{ env rewrite } \text{plusZero } n = \text{env}
\]

\[
\text{initByK} \{n\} \text{ (Succ } k \text{) env rewrite } \text{plusSucc } n \text{ k} = \text{init } (\text{initByK } k \text{ env})
\]

Again, we nest \(k\) applications of \(\text{snoc}\) and \(\text{init}\) and use the proofs \(\text{plusZero}\) and \(\text{plusSucc}\) to rewrite the types. Also note that the order of nesting is different, which is required when we define proofs regarding these functions.

Given these functions, we define \(\text{shrink}\) and \(\text{grow}\) as follows:

\[
\text{shrink} : \text{forall } \{n k : \text{Nat}\} \rightarrow
\]

\[
(u : \text{U } n) \rightarrow
\]

\[
(\text{forall } \{\text{env} : \text{Env } (n + k)\} \rightarrow \text{elU } (\text{weakenByK } k \text{ u env}) \rightarrow
\]

\[
(\text{forall } \{\text{env} : \text{Env } n\} \rightarrow \text{elU } u \text{ env})
\]

\[
\text{shrink } \{n\} \{k\} u f \{\text{env}\} = \text{shrink}' (f \{\text{snocByK } k \text{ env}\})
\]

where

\[
\text{shrink}' : \text{elU } (\text{weakenByK } k \text{ u} (\text{snocByK } k \text{ env}) \rightarrow \text{elU } u \text{ env}
\]

\[
\text{shrink}' \times \text{rewrite} \text{ shrinkableByK } k \text{ u env} = x
\]

\[
\text{grow} : \text{forall } \{n k : \text{Nat}\} \rightarrow
\]

\[
(u : \text{U } n) \rightarrow
\]

\[
(\text{forall } \{\text{env} : \text{Env } n\} \rightarrow \text{elU } u \text{ env}) \rightarrow
\]

\[
(\text{forall } \{\text{env} : \text{Env } (n + k)\} \rightarrow \text{elU } (\text{weakenByK } k \text{ u env})
\]

\[
\text{grow } \{n\} \{k\} u f \{\text{env}\} = \text{grow}' (f \{\text{initByK } k \text{ env}\})
\]

where

\[
\text{grow}' : \text{elU } u \text{ (initByK } k \text{ env}) \rightarrow \text{elU } (\text{weakenByK } k \text{ u env}
\]

\[
\text{grow}' \times \text{rewrite} \text{ growableByK } k \text{ u env} = x
\]

\[\text{2} \text{We could have chosen any representation as the stand-in value since } \text{init } (\text{snoc } \text{ env}) \equiv \text{env}, \text{ meaning that adding a stand-in value and dropping it again results in the same environment.}\]
Both functions supply either a larger or a smaller environment to the given function. We need the following two proofs on the behaviour of `snocByK` and `initByK`:

\[
\text{shrinkableByK} : \forall \{n : \text{Nat}\} \rightarrow (k : \text{Nat}) \rightarrow (u : U_n) \rightarrow (\text{env} : \text{Env} n) \rightarrow \\
\quad \text{elU} (\text{weakenByK} k u) (\text{snocByK} k \text{env}) \equiv \text{elU} u \text{env}
\]

\[
\text{growableByK} : \forall \{n : \text{Nat}\} \rightarrow (k : \text{Nat}) \rightarrow (u : U_n) \rightarrow (\text{env} : \text{Env} (n + k)) \rightarrow \\
\quad \text{elU} u (\text{initByK} k \text{env}) \equiv \text{elU} (\text{weakenByK} k u) \text{env}
\]

The first states that a value of a weakened representation with a larger environment containing stand-in values is equal to a value of the original representation and environment. The second states that a value with a smaller environment is equal to a value of the weakened representation where the stand-in values are still present. These are defined by induction on \(k\) and use the following proofs on shrinking and growing with a single element:

\[
\text{shrinkable} : \forall \{n : \text{Nat}\} \rightarrow \\
\quad (u : U_n) \rightarrow (\text{env} : \text{Env} n) \rightarrow \\
\quad \text{elU} (\text{weaken} u) (\text{snoc env}) \equiv \text{elU} u \text{env}
\]

\[
\text{growable} : \forall \{n : \text{Nat}\} \rightarrow \\
\quad (u : U_n) \rightarrow (\text{env} : \text{Env} (\text{Succ} n)) \rightarrow \\
\quad \text{elU} u (\text{init} \text{env}) \equiv \text{elU} (\text{weaken} u) \text{env}
\]

These functions are straightforwardly defined by induction on \(u\). The complete definitions of these proofs can be found in Appendix 6.C.1.

### 6.5.2 Instantiation

The next step in the cast function is to instantiate the representation found in the dynamic value using the substitution obtained from the instance-of algorithm. To this end, we define the following function:

\[
\text{instantiate} \quad \forall \{n : \text{Nat}\} \rightarrow \\
\quad (u : U_n) \rightarrow (\text{subst} : \text{Subst} n n) \rightarrow \\
\quad (\text{forall} \{\text{env} : \text{Env} n\} \rightarrow \text{elU} u \text{env}) \rightarrow \\
\quad (\text{forall} \{\text{env} : \text{Env} n\} \rightarrow \text{elU} (\text{apply subst} u) \text{env})
\]

\[
\text{instantiate} \quad u \text{ subst} \ f \ \{\text{env}\} \\
\text{ rewrite} \ \text{mergeCorrect} \ u \ \text{subst} \ \text{env} = f \ \{\text{merge subst} \ \text{env}\}
\]

It is important to observe that this function delivers a value of a representation where the substitution is applied, this allows us later in the cast function to deploy the proof of correctness of the instance-of algorithm as defined in Section 6.5.2. The idea behind this function is that the value found in the dynamic value takes

---

3Whenever we give a proof of equality on the result of interpreting representations using \(\text{elU}\), we compare types which themselves have type \(\text{Set}_1\). However, as defined in Section 6.3.1, the \(\equiv\) operator takes two values of type \(\text{Set}\). We implicitly use universe polymorphism (Harper and Pollack, 1989) to generalise this operator to take two values of \(\text{Set}_i\) for any level \(i\).
an environment as an implicit argument. Hence, we can instantiate variables by merging the information from the substitution in this environment. The function \texttt{merge} takes care of this:

\[
\text{merge} : \forall \{n : \text{Nat}\} \to \text{Subst } n n \to \text{Env } n \to \text{Env } n
\]

\[
\text{merge subst env} = \text{mergeClose subst env env}
\]

We need another function \texttt{mergeClose} that decouples the type of the values in the substitution and its length such that we can perform recursion, much like the function \texttt{findInSubst} from Section 6.4.1. It takes an additional argument to close values from the substitution when they are included in the environment:

\[
\text{mergeClose} : \forall \{n m : \text{Nat}\} \to
\]

\[
\text{Subst } m n \to \text{Env } n \to \text{Env } m \to \text{Env } n
\]

\[
\text{mergeClose }\{\text{Zero}\} = \text{Nil}
\]

\[
\text{mergeClose }\{\text{Succ } n\} (\text{Cons Nothing subst}) (\text{Cons } u \text{ env} \text{ env}') =
\]

\[
\text{Cons } u (\text{mergeClose subst env env'}
\]

\[
\text{mergeClose }\{\text{Succ } n\} (\text{Cons } (\text{Just } u) \text{ subst}) (\text{Cons } \_ \text{ env} \text{ env}') =
\]

\[
\text{Cons } (\text{close } u \text{ env'}) (\text{mergeClose subst env env'})
\]

This function zips the substitution and the environment together by performing induction on their length. When there is no substitution information present, we take the information from the environment. Otherwise, we close the value and replace this position in the environment, after which we continue recursion. The function \texttt{close} gives us a representation that does not contain any variables:

\[
\text{close} : \forall \{n : \text{Nat}\} \to \text{U } n \to \text{Env } n \to \text{U } \text{ Zero}
\]

\[
\text{close } \text{NAT} = \text{NAT}
\]

\[
\text{close } (\text{PAIR } u u') \text{ env} = \text{PAIR } (\text{close } u \text{ env}) (\text{close } u' \text{ env})
\]

\[
\text{close } (u \Rightarrow u') \text{ env} = \text{close } u \text{ env} \Rightarrow \text{close } u' \text{ env}
\]

\[
\text{close } (\text{VAR } i) \text{ env} = \text{findInEnv } i \text{ env}
\]

Note that this function mimics the behaviour of the interpretation function \texttt{elU0} from Section 6.3.2.

Then, having merged the information from the substitution into the environment, we need to give a proof of correctness that we indeed deliver a value that fits the proof of correctness of the instance-of algorithm:

\[
\text{mergeCorrect} : \forall \{n : \text{Nat}\} \to
\]

\[
(u : \text{U } n) \to (\text{subst : Subst } n n) \to (\text{env : Env } n) \to
\]

\[
\text{elU } (\text{apply subst } u) \text{ env} \equiv \text{elU } u (\text{merge subst env})
\]

This proof captures the relation between substitutions and environments. Namely, they behave the same: it does not matter whether one instantiates variables by applying that substitution to a representation, or merges the types to which variables refer in a substitution with an environment. The proof proceeds by induction on \(u\). In the case of a variable it distinguishes cases by the presence of a value for that reference in the substitution, just like we do in merging environments and substitutions. The complete definition of this proof can be found in Appendix 6.C.2.
6.5.3 Putting it all together

Now it is time to put all of it together into a single cast function:

\[
\begin{align*}
\text{cast} & : (v : V) \rightarrow \text{Dyn} \rightarrow \text{Maybe} (\text{el} V v) \\
\text{cast} & (\text{FORALL} \{n\} v \rightarrow (\text{dyn} (\text{FORALL} \{m\} m) \rightarrow) \text{with} \text{cmp} n m \\
\text{cast} & (\text{FORALL} \{n\} u1 \rightarrow (\text{dyn} (\text{FORALL} \{.n\} u2) f) \rightarrow) \text{EQ} \\
& \quad \text{with} \text{iofCorrectness} u1 u2 \\
& \quad \text{Just} (\text{Witness subst eq}) = \text{Just} (\text{coerce eq (instantiate u2 subst f)}) \\
& \quad \text{Nothing} \\
\text{cast} & (\text{FORALL} u1 \rightarrow (\text{dyn} (\text{FORALL} u2) f) \rightarrow) \text{LT k} \\
& \quad \text{with} \text{iofCorrectness} (\text{weakenByK} k u1) u2 \\
& \quad \text{Just} (\text{Witness subst eq}) = \\
& \quad \text{Just} (\text{shrink u1 (coerce eq (instantiate u2 subst f)))} \\
& \quad \text{Nothing} \\
\text{cast} & (\text{FORALL} u1 \rightarrow (\text{dyn} (\text{FORALL} u2) f) \rightarrow) \text{GT k} \\
& \quad \text{with} \text{iofCorrectness} u1 (\text{weakenByK} k u2) \\
& \quad \text{Just} (\text{Witness subst eq}) = \\
& \quad \text{Just} (\text{coerce eq (instantiate (weakenByK k u2) subst (grow u2 f)))} \\
& \quad \text{Nothing}
\end{align*}
\]

First, we compare the number of variables that both representations use using the function `cmp` as defined in Section 6.3.2. If they can use the same number of variables, we use `iofCorrectness` to possibly obtain a substitution that turns `u2` into `u1`, witnessed by a proof of correctness. Next, we instantiate the type of the value `f` using this substitution and coerce the value using its corresponding proof.

If the provided representation can use fewer variables than the representation in the dynamic value, we weaken the first argument before we use `iofCorrectness`. Then, again, we instantiate the type of the value `f` and coerce it using the proof of correctness. Since we weakened `u1`, we need to shrink the resulting value.

In the final case, we weaken the second argument before we use `iofCorrectness`. Consequently, instantiation takes place on the weakened version of `u2` and we have to grow the corresponding value as well. Then, we use the proof of correctness to coerce the resulting value.

Coercion of a value is defined by the following function:

\[
\begin{align*}
do coerc & : \forall n : \text{Nat} \forall u1 u2 : \text{U} n \rightarrow \\
& \quad u1 \equiv u2 \rightarrow \\
& \quad (\forall \text{env} : \text{Env} n \rightarrow \text{elU u1 env}) \rightarrow \\
& \quad (\forall \text{env} : \text{Env} n \rightarrow \text{elU u2 env}) \\
\text{coerce} \text{Refl f} & = f
\end{align*}
\]

We deploy the argument equality proof by pattern matching `Refl`, allowing us to return `f`. This concludes the definition of the cast function.

6.6 Related work

Dynamic typing in Haskell has been studied by both Baars and Swierstra (2002) and Cheney and Hinze (2002) around the same time. Both approaches only con-
6.6 Related work

sidered monomorphic dynamic typing. Respectively, they state: “Whether our approach can easily be extended with dynamic polymorphism is as yet unknown and a subject of further research.” and “We believe our Dynamic also can support making values of closed polymorphic types dynamic, although we have yet to experiment with unifying and pattern-matching polymorphic type representations.”. A similar but weaker research question has been formulated by Sheard et al. (2005) and said to be difficult (Sheard and Pasalić, 2008): “Is it possible to build [...] singleton types to represent polymorphic types? While we have tried many approaches we are not yet satisfied with the generality of any of them.”. Unfortunately, there has not been any follow up on this work and these research questions have neither been proven nor disproven by the authors. In this chapter we discuss the difficulties in representing polymorphic types using generalised algebraic datatypes in Haskell. We also argue that a universe for the representation of types and its interpretation needs to be separated to embed polymorphic dynamic typing in a functional language.

A workaround in Haskell to support dynamic typing with polymorphism has been suggested by Pang et al. (2004). The idea is that any polymorphic value can be made monomorphic by wrapping it in a vanilla datatype. While this allows us to move around such dynamic values, we are not able to unwrap it with a less general type by an instance-of algorithm, like we describe in this chapter.

There has also been some work on extending the Haskell library for monomorphic dynamic typing with polymorphism (Stewart, 2010). There it is argued, as we do in this chapter, that polymorphism in representations requires some kind of unification. Instead of supporting this via a library, or by extending the language itself, a hook to the compiler is provided to invoke a unification mechanism at run time. In our approach we do not follow this path but investigate the embedding in a language itself, thereby also experimenting with and learning about the expressivity of the language and its features.

The combination of dynamic typing and dependently-typed programming is not entirely new. Ou et al. (2004) argue that a programmer needs fine-grained control over the number of type annotations and the level of compile-time safety. A new system is described where pieces of the program are either marked dependent or simple, where the latter case is verified at run time. However, our goal is different in that we consider the embedding of dynamic typing in a dependently-typed language via a universe and its interpretation, instead of completely merging the two idioms by extending the system itself.

We use a dependently-typed language mostly for its ability to separate a universe from its interpretation, such that we can compare representations. Crary and Weirich (1999) use the same approach and define interpretation functions on a universe for the representation of polymorphic types, very much like our interpretations. However, their work concerns a separate system named LX that is completely dedicated to analyses of types within a programming language, whereas we consider the embedding of such analyses in an already existing language.

A universe of representations and their interpretation functions has been shown to be an effective approach in generic programming in a dependently-typed setting (Altenkirch and McBride, 2003; Oury and Swierstra, 2008). Also, the duality relation between generic programming and dynamic typing has been described
earlier (Cheney and Hinze, 2002). Hence, it comes as no surprise that we can use universe construction for dynamic typing as well. However, to our knowledge we are the first to investigate this relation in the context of the embedding of polymorphic dynamic typing.

6.7 Conclusion

We have explored the embedding of polymorphic dynamic typing as a library in a functional language. We argued that an approach in a functional language like Haskell requires generalised algebraic datatypes to relate values to the representation of their types, but extending the embedding of monomorphic types to polymorphic types presents difficulties. Type representations would need to carry additional administration in the form of an environment. This closes the door on their comparison since equal references from different representations involve different environments.

In essence, we have shown that a universe for the representation of types needs to be separated from its interpretation as a type. While there are some possibilities to perform this separation in Haskell by making heavy use of generalised algebraic datatypes and type families, we believe that a more natural approach is offered by a dependently-typed language such as Agda. There, we are able to elegantly postpone attaching meaning to a representation until after performing any comparison.

We first defined a framework for polymorphic dynamic typing in Agda with structural equality of representations. Then, we extended this approach with an instance-of algorithm and gave a complete proof of its correctness in Agda itself. Finally, we defined a single cast function that aligns, instantiates, and coerces representations and values, such that we can unwrap a value from a dynamic value by offering a less general or equal type representation.

In hindsight, our choice of a universe might seem to contain redundancy since pairs and functions are treated the same by every function that works on representations. However, this apparent redundancy highlights a nice property of our approach: values are abstract and we only deal with the representations of their types. This prevents us for example from having to deal with co- and contravariance in the case of function values.

An interesting opportunity for future work is to see if the approach in Agda can be transferred to Haskell. Now that we have defined a complete framework, the parts are clear where the additional power of a dependently-typed language is most used. We expect that this knowledge will give great insight in what is needed to perform an embedding in Haskell.
6.A Proof prelude

sym : forall {a : Set} {x y : a} -> x ≡ y -> y ≡ x
sym Refl = Refl

trans : forall {a : Set} {x y z : a} -> x ≡ y -> y ≡ z -> x ≡ z
trans Refl Refl = Refl

cong : forall {a b : Set} {x y : a} -> (f : a -> b) -> x ≡ y -> f x ≡ f y
cong Refl Refl = Refl

magic : forall {a : Set} -> Empty -> a
magic ()

justInj : forall {a : Set} {x y : a} -> Just x ≡ Just y -> x ≡ y
justInj Refl = Refl

justNothing : forall {a : Set} {x : a} -> Nothing ≡ Just x -> Empty
justNothing ()

pairInj : forall {n : Nat} {u1 u1' u2 u2' : U n} ->
  PAIR u1 u1' ≡ PAIR u2 u2' ->
  Pair (u1 ≡ u2) (u1' ≡ u2')
pairInj Refl Refl = Refl, Refl

funInj : forall {n m : Nat} {u1 u1' u2 u2' : U n} ->
  (u1 ⇒ u1') ≡ (u2 ⇒ u2') ->
  Pair (u1 ≡ u2) (u1' ≡ u2')
funInj Refl Refl = Refl, Refl

varInj : forall {n : Nat} {i j : Fin n} -> VAR i ≡ VAR j -> i ≡ j
varInj Refl = Refl

fsInj : forall {n : Nat} {i j : Fin n} -> Fs i ≡ Fs j -> i ≡ j
fsInj Refl = Refl

fsNeqInj : forall {n : Nat} -> (i j : Fin n) -> Fs i ≠ Fs j -> i ≠ j
fsNeqInj Fz Fz neq = magic (neq Refl)
fsNeqInj (Fs _) _ = λ ()
fsNeqInj (Fs _) Fz _ = λ ()
fsNeqInj (Fs _) (Fs _) neq = λ eq -> neq (cong Fs eq)

consInj : forall {n m : Nat} {mu mu' : Maybe (U m)}
  {subst subst' : Subst m n} ->
  Cons mu subst ≡ Cons mu' subst' ->
  Pair (mu ≡ mu') (subst ≡ subst')
consInj Refl Refl = Refl, Refl
6 Embedding polymorphic dynamic typing

6.B Proofs regarding instance-of algorithm

6.B.1 Substitution

\[ \text{decU'} : \forall \{ n : \text{Nat} \} \to (u1 \, u2 : U \, n) \to \text{Either} \ (u1 \equiv u2) \ (u1 \not\equiv u2) \]

\[ \text{decU'} \ \text{NAT} \ \text{NAT} = \text{Inl Refl} \]

\[ \text{decU'} \ \text{NAT} \ \text{(PAIR } \_ \_ \text{)} = \text{Inr} \ (\lambda ()) \]

\[ \text{decU'} \ \text{NAT} \ (\_ \Rightarrow \_) = \text{Inr} \ (\lambda ()) \]

\[ \text{decU'} \ \text{NAT} \ \text{(VAR } \_) = \text{Inr} \ (\lambda ()) \]

\[ \text{decU'} \ (\text{PAIR } \_ \_ ) \ \text{NAT} = \text{Inr} \ (\lambda ()) \]

\[ \text{decU'} \ (\text{PAIR } u1 \, u1') \ (\text{PAIR } u2 \, u2') \text{ with } \text{decU'} \ u1 \, u2 \mid \text{decU'} \ u1' \, u2' \]

\[ \text{decU'} \ (\text{PAIR } u1 \, u1') \ (\text{PAIR } .u1 \,.u1') \mid \text{Inl Refl} \mid \text{Inl Refl} = \text{Inl Refl} \]

\[ \text{decU'} \ (\text{PAIR } u1 \, u1') \ (\text{PAIR } .u1 \,.u1') \mid \text{Inl Refl} \mid \text{Inr neq} = \]

\[ \text{Inr} \ (\lambda eq \to \text{neq} \ (\text{snd} \ (\text{pairInj eq}))) \]

\[ \text{decU'} \ (\text{PAIR } \_ \_ ) \ (\_ \Rightarrow \_) = \text{Inr} \ (\lambda ()) \]

\[ \text{decU'} \ (\text{PAIR } \_ \_ ) \ (\_ \Rightarrow \_) = \text{Inr} \ (\lambda ()) \]

\[ \text{decU'} \ _ \Rightarrow \_) \ (\_ \Rightarrow \_) = \text{Inr} \ (\lambda ()) \]

\[ \text{decU'} \ (u1 \Rightarrow u1') \ (u2 \Rightarrow u2') \text{ with } \text{decU'} \ u1 \, u2 \mid \text{decU'} \ u1' \, u2' \]

\[ \text{decU'} \ (u1 \Rightarrow u1') \ (\_ \Rightarrow \_) \mid \text{Inl Refl} \mid \text{Inl Refl} = \text{Inl Refl} \]

\[ \text{decU'} \ (u1 \Rightarrow u1') \ (\_ \Rightarrow \_) \mid \text{Inl Refl} \mid \text{Inr neq} = \]

\[ \text{Inr} \ (\lambda eq \to \text{neq} \ (\text{snd} \ (\text{funInj eq}))) \]

\[ \text{decU'} \ (\_ \Rightarrow \_) \ (\_ \Rightarrow \_) = \text{Inr} \ (\lambda ()) \]

\[ \text{decU'} \ (\_ \Rightarrow \_) \ (\_ \Rightarrow \_) = \text{Inr} \ (\lambda ()) \]

\[ \text{decU'} \ (\_ \Rightarrow \_) \ (\_ \Rightarrow \_) = \text{Inr} \ (\lambda ()) \]

\[ \text{decU'} \ (\text{VAR } i) \ (\text{VAR } j) \text{ with } \text{decFin'} \ i \, j \]

\[ \text{decU'} \ (\text{VAR } i) \ (\text{VAR } .i) \mid \text{Inl Refl} = \text{Inl Refl} \]

\[ \text{...} \mid \text{Inr neq} = \text{Inr} \ (\lambda eq \to \text{neq} \ (\text{funInj eq})) \]

\[ \text{decFin'} : \forall \{ n : \text{Nat} \} \to (i \, j : \text{Fin } n) \to \text{Either} \ (i \equiv j) \ (i \not\equiv j) \]

\[ \text{decFin'} \ Fz \ Fz = \text{Inl Refl} \]

\[ \text{decFin'} \ Fz \ (\text{Fs } i) = \text{Inr} \ (\lambda ()) \]

\[ \text{decFin'} \ (\text{Fs } i) \ Fz = \text{Inr} \ (\lambda ()) \]

\[ \text{decFin'} \ (\text{Fs } i) \ (\text{Fs } j) \text{ with } \text{decFin'} \ i \, j \]

\[ \text{decFin'} \ (\text{Fs } i) \ (\text{Fs } .i) \mid \text{Inl Refl} = \text{Inl Refl} \]

\[ \text{...} \mid \text{Inr neq} = \text{Inr} \ (\lambda eq \to \text{neq} \ (\text{funInj eq})) \]

148
6.B.2 Accumulating instance-of function

\[
\begin{align*}
iofCorrect & : \forall \{ n : \text{Nat} \} \to \\
& \quad (u1 u2 : U n) \to (\text{subst subst'} : \text{Subst } n n) \to \\
& \quad \text{iofAcc u1 u2 subst } \equiv \text{Just subst'} \to \\
& \quad \text{apply subst'} u2 \equiv u1 \\
\end{align*}
\]

with inspect (iofAcc u1 u2 subst)

... | Just subst" with \( \equiv \) eq' rewrite eq' | iofCorrect u1' u2' subst" subst' eq

|  | iofStable u1 u2 u1' u2' subst subst" subst' eq' eq

|  | iofCorrect u1 u2 subst subst" subst' eq'

|  | iofStable u1' u2' subst subst' subst eq'

|  | iofStable subst subst' subst eq'

... | Nothing with \( \equiv \) eq' rewrite eq' = magic (justNothing eq)

iofStable : \forall \{ n : \text{Nat} \} \to \\
\quad (u1 u2 u1' u2' : U n) \to (\text{subst subst'} : \text{Subst } n n) \to \\
\quad \text{iofAcc u1 u2 subst } \equiv \text{Just subst'} \to \\
\quad \text{iofAcc u1' u2' subst' } \equiv \text{Just subst" } \to \\
\quad \text{apply subst" u2 } \equiv \text{apply subst' u2' }

with inspect (iofAcc u1 u2 subst)

... | Just subst" with \( \equiv \) eq'' rewrite eq'' | iofStable u1' u2' u' subst subst'' subst' subst eq'' eq eq'

|  | cong (\( \lambda u'' \to \text{PAIR } u'' \text{ (apply subst' u2')} \)) (\text{trans p (sym q)})

iofStable NAT \quad \text{NAT } \quad \_ \quad \_ \quad \_ \quad \_ = \text{Refl}

iofStable NAT \quad (\text{PAIR } \_ \_ ) \quad \_ \quad \_ \quad ()

iofStable \quad (\_ \Rightarrow \_ ) \quad \_ \quad \_ \quad ()

iofStable \quad (\_ \Rightarrow \_ ) \quad \_ \quad \_ \quad ()

iofStable (PAIR \_ ) \quad \_ \quad \_ \quad ()

iofStable (PAIR \_ ) \quad \_ \quad \_ \quad ()

iofStable (PAIR u1 u1') (PAIR u2 u2') subst subst' eq

rewite iofVarStep u i subst | findable u i subst subst' eq = Reff

iofStable NAT \quad \text{NAT } \quad \_ \quad \_ \quad \_ \quad \_ = \text{Refl}

iofStable (PAIR \_ \_ ) \quad \_ \quad \_ \quad \_ \quad ()

iofStable (PAIR \_ \_ ) \quad \_ \quad \_ \quad ()

iofStable (PAIR \_ \_ ) \quad \_ \quad \_ \quad ()

iofStable (PAIR \_ \_ ) \quad \_ \quad \_ \quad ()

iofStable (PAIR u1 u1') (PAIR u2 u2') subst subst' subst eq eq'

with inspect (iofAcc u1 u2 subst)

... | Just subst" with \( \equiv \) eq'' rewrite eq'' | iofStable u1' u2' u' subst subst'' subst' subst eq'' eq eq'

|  | cong (\( \lambda u'' \to \text{PAIR } u'' \text{ (apply subst' u2')} \)) (\text{trans p (sym q)})
where

lemma : iofAcc (PAIR u1' u) (PAIR u2' u') subst'' ≡ Just subst''
lemma with inspect (iofAcc u1' u2' subst'')
  |
  | Just _ with≡ _ rewrite eq | eq' = Refl
  | Nothing with≡ eq'' = magic (justNothing (trans (sym eq'')) eq))
p : apply subst'' u2 ≡ apply subst'' u2
p = iofStable u1 u2 (PAIR u1' u) (PAIR u2' u') subst'' subst'' subst eq'' lemma
q : apply subst' u2 ≡ apply subst'' u2
q = iofStable u1 u2 u1' u2' subst subst'' subst' subst' eq'' eq'
  with inspect (iofAcc u1 u2 subst)
  |
  | Just subst'' with≡ eq'' rewrite eq'' | iofStable u1' u2' u u' subst'' subst' subst'' subst eq' eq' =
  | cong (λ u'' → u'' ⇒ apply subst' u2') (trans p (sym q))

where

lemma : iofAcc (u1' ⇒ u) (u2' ⇒ u') subst'' ≡ Just subst''
lemma with inspect (iofAcc u1' u2' subst'')
  |
  | Just _ with≡ _ rewrite eq | eq' = Refl
  | Nothing with≡ eq'' = magic (justNothing (trans (sym eq'')) eq))
p : apply subst'' u2 ≡ apply subst'' u2
p = iofStable u1 u2 (PAIR u1' ⇒ u) (PAIR u2' ⇒ u') subst subst'' subst' subst eq'' lemma
q : apply subst' u2 ≡ apply subst'' u2
q = iofStable u1 u2 u1' u2' subst subst'' subst' subst' eq'' eq'
  with inspect (iofAcc u1 u2 subst)
  |
  | Just subst'' with≡ eq'' rewrite eq'' | iofStable u1' u2' u u' subst subst'' subst' subst'' subst eq' eq' =
  | cong (λ u'' → u'' ⇒ apply subst' u2') (trans p (sym q))

iofStable (PAIR _ _) (_ ⇒ _) _ _ _ _ () _
iofStable (_ ⇒ _) NAT _ _ _ _ _ () _
iofStable (_ ⇒ _) (PAIR _ _) _ _ _ _ _ () _
iofStable (u1 ⇒ u1') (u2 ⇒ u2') u u' subst subst' subst'' subst eq'' eq'
  rewrite iofVarStep u1 i subst with findable u1 i subst subst' eq
  |
  | eq'' rewrite eq'' | findStable u1 u u' i subst subst'' subst eq'' eq' = Refl

iofVarStep : forall {n : Nat} →
  (u : U n) → (i : Fin n) → (subst : Subst n n) →
  iofAcc u (VAR i) subst ≡ update i u subst
iofVarStep NAT _ _ ≡ Refl
iofVarStep (PAIR _ _) _ _ ≡ Refl
iofVarStep (_ ⇒ _) _ _ ≡ Refl
iofVarStep (VAR _ _) _ _ ≡ Refl

findable : forall {n m : Nat} →
  (u : U m) → (i : Fin n) → (subst subst' : Subst m n) →
  update i u subst ≡ Just subst' →
  findInSubst i subst subst' ≡ Just u
findable_ Fz  (Cons Nothing _) (Cons _ _) eq
  rewrite fst (consInj (justInj eq)) = Refl
findable u Fz  (Cons (Just u') _) (Cons _ _) eq with decU' u u'
findable u Fz  (Cons (Just .u) _) (Cons _ _) eq | Inl Refl
  rewrite fst (consInj (justInj (sym eq))) = Refl
findable_ Fz  (Cons (Just _) _) (Cons _ _) () | Inr _
findable u (Fs i) (Cons _ subst) (Cons _ subst') eq
  with inspect (update i u subst)
... | Just _ with≡ eq' rewrite eq' | snd (consInj (justInj eq))
  | findable u i subst subst' eq' = Refl
... | Nothing with≡ eq' rewrite eq' = magic (justNothing eq)

findStable : forall {n : Nat} →
  (u u1 u2 : U n) → (i : Fin n) → (subst subst' : Subst n n) →
  findInSubst i subst ≡ Just u →
  iofAcc u u1 u2 subst ≡ Just subst' →
  findInSubst i subst' ≡ Just u
findStable _ NAT  NAT _ _ _ _ eq eq'
  rewrite justInj eq' | eq = Refl
findStable _ (PAIR _) NAT _ _ _ _ ()
findStable _ (_ ⇒ _) NAT _ _ _ _ ()
findStable _ (PAIR _) NAT _ _ _ _ ()
findStable u (PAIR u1 u1') (PAIR u2 u2') i subst subst' eq eq'
  with inspect (iofAcc u1 u2 subst)
... | Just subst'' with≡ eq''
  rewrite eq'' | findStable u1' u1' i subst'' subst''
  (findStable u1 u2 i subst subst'' eq eq'') eq' = Refl
... | Nothing with≡ eq'' rewrite eq'' = magic (justNothing eq'')
findStable _ (PAIR _) (_ ⇒ _) _ _ _ _ ()
findStable _ (_ ⇒ _) NAT _ _ _ _ ()
findStable _ (_ ⇒ _) (PAIR _) _ _ _ _ ()
findStable u (u1 ⇒ u1') (u2 ⇒ u2') i subst subst' eq eq'
  with inspect (iofAcc u1 u2 subst)
... | Just subst'' with≡ eq''
  rewrite eq'' | findStable u1' u1' i subst'' subst''
  (findStable u1 u2 i subst subst'' eq eq'') eq' = Refl
... | Nothing with≡ eq'' rewrite eq'' = magic (justNothing eq'')
findStable _ (VAR _) NAT _ _ _ _ ()
findStable _ (VAR _) (PAIR _) _ _ _ _ ()
findStable _ (VAR _) (_ ⇒ _) _ _ _ _ ()
findStable u u1 (VAR j) i subst subst' eq eq'
  rewrite iofVarStep u1 j subst with decFin' i j | decU' u u1
findStable u .u (VAR .i) i subst subst' _ eq'
  | Inl Refl | Inl Refl rewrite findable u i subst subst' eq' = Refl
findStable u u1 (VAR .i) i subst subst' eq eq'
  | Inl Refl | Inr neq = magic (neq (updateEq u1 i subst subst' eq eq'))
... | Inr neq | _ rewrite updateNeq u1 i j subst subst' neq eq eq' = Refl
updateEq : \forall \{ n m : \text{Nat} \} \to \\
  (u u' : U m) \to (i : \text{Fin n}) \to (\text{subst} \text{subst'} : \text{Subst} m n) \to \\
  \text{findInSubst} i \text{subst} \equiv \text{Just} u \to \\
  \text{update} i u' \text{subst} \equiv \text{Just} \text{subst'} \to u \equiv u'

updateEq Fz (\text{ConsNothing}) (\text{ConsNothing}) ()
updateEq u' Fz (\text{ConsJust u''}) (\text{ConsNothing}) eq' with decU' u' u''
updateEq u' Fz (\text{ConsJust .u'}) (\text{ConsNothing}) eq eq'

rewrite justInj (\text{sym eq}) = \text{Refl}
updateEq Fz (\text{ConsNothing}) (\text{ConsNothing}) () | \text{Inl Refl}
updateEq u (\text{Fs i}) (\text{ConsNothing}) (\text{ConsNothing}) eq eq'

with inspect (update i u' subst)
... | Just subst'' with-≡ eq'' rewrite eq''
  | updateEq u u' subst subst'' eq eq' = \text{Refl}
... | Nothing with-≡ eq'' rewrite eq'' = \text{magic (justNothing eq')}

updateNeq : \forall \{ n m : \text{Nat} \} \to \\
  (u u' : U m) \to (i j : \text{Fin n}) \to (\text{subst} \text{subst'} : \text{Subst} m n) \to \\
  i \neq j \to \\
  \text{findInSubst} i \text{subst} \equiv \text{Just} u \to \\
  \text{update} j u' \text{subst} \equiv \text{Just} \text{subst'} \to \\
  \text{findInSubst} i \text{subst'} \equiv \text{Just} u

updateNeq Fz Fz (\text{ConsNothing}) _ _ neq _ _ = \text{magic (neq Refl)}
updateNeq (\text{Fs _}) Fz (\text{ConsNothing}) (\text{ConsNothing}) _ _ ()
updateNeq (\text{Fs _}) Fz (\text{ConsNothing}) (\text{ConsNothing}) (\text{ConsJust _ _}) _ _ eq' eq''

rewrite snd (\text{consInj (justInj eq'')) | eq' = \text{Refl}
updateNeq u' Fz (\text{ConsNothing}) (\text{ConsNothing}) _ _ eq''

with decU' u' u''
updateNeq u' Fz (\text{ConsJust u''}) _ _ eq''
  | \text{Inl Refl rewrite justInj eq'' | eq' = \text{Refl}
updateNeq Fz Fz (\text{ConsNothing}) (\text{ConsNothing}) _ _ ()
  | \text{Inr _}
updateNeq u' Fz (\text{Fs j}) (\text{ConsNothing}) _ _ eq''

with update j u' subst
... | Just _ rewrite eq' | fst (\text{consInj (justInj (sym eq''))}) = \text{Refl}
updateNeq Fz Fz (\text{ConNothing}) (\text{ConNothing}) _ _ ()
  | Nothing
updateNeq u' (\text{Fs i}) (\text{Fs j}) (\text{ConNothing}) (\text{ConNothing}) (\text{ConNothing}) _ _ neq eq' eq''

with inspect (update j u' subst)
... | Just _ with-≡ eq''' rewrite eq''' | snd (\text{consInj (justInj eq'''))
  | updateNeq u u' i j subst subst' (\text{fsNeqInj i j neq}) eq' eq''' = \text{Refl}
... | Nothing with-≡ eq''' rewrite eq''' = \text{magic (justNothing eq''')}

152
6.C Proofs regarding cast

6.C.1 Alignment

plusZero : (n : Nat) → n + Zero ≡ n
plusZero Zero = Refl
plusZero (Succ n) rewrite cong Succ (plusZero n) = Refl

plusSucc : (n m : Nat) → n + Succ m ≡ Succ (n + m)
plusSucc Zero _ = Refl
plusSucc (Succ n) m rewrite cong Succ (plusSucc n m) = Refl

shrinkableByK : forall {n : Nat} →
   (k : Nat) → (u : U n) → (env : Env n) →
   elU (weakenByK k u) (snocByK k env) ≡ elU u env
shrinkableByK {n} Zero _ _ rewrite plusZero n = Refl
shrinkableByK {n} (Succ k) u env
   rewrite plusSucc n k | shrinkableByK (weaken u) (snoc env)
   | shrinkable u env = Refl

growableByK : forall {n : Nat} →
   (k : Nat) → (u : U n) → (env : Env (n + k)) →
   elU u (initByK k env) ≡ elU (weakenByK k u) env
growableByK {n} Zero _ _ rewrite plusZero n = Refl
growableByK {n} (Succ k) u env
   rewrite plusSucc n k | growable u (initByK k env)
   | growableByK k (weaken u) env = Refl

shrinkable : forall {n : Nat} →
   (u : U n) → (env : Env n) →
   elU (weaken u) (snoc env) ≡ elU u env
shrinkable NAT _ _ = Refl
shrinkable (PAIR u u') env rewrite shrinkable u env | shrinkable u' env = Refl
shrinkable (u ⇒ u') env rewrite shrinkable u env | shrinkable u' env = Refl
shrinkable (VAR Fz) (Cons _) = Refl
shrinkable (VAR (Fs i)) (Cons _ env) rewrite shrinkable (VAR i) env = Refl

growable : forall {n : Nat} →
   (u : U n) → (env : Env (Succ n)) →
   elU u (init env) ≡ elU (weaken u) env
growable NAT _ _ = Refl
growable (PAIR u u') env rewrite growable u env | growable u' env = Refl
growable (u ⇒ u') env rewrite growable u env | growable u' env = Refl
growable (VAR Fz) (Cons _ (Cons _)) = Refl
growable (VAR (Fs i)) (Cons _ (Cons u' env))
   rewrite growable (VAR i) (Cons u' env) = Refl
6. C. 2 Instantiation

\texttt{mergeCorrect :: forall \{n : Nat\} \rightarrow (u : U n) \rightarrow (\text{subst} : \text{Subst n n}) \rightarrow (\text{env} : \text{Env n}) \rightarrow \text{elU} (\text{apply subst u}) \text{ env} \equiv \text{elU} u (\text{merge subst env})}

\texttt{mergeCorrect NAT \_ \_ = \text{Refl}}

\texttt{mergeCorrect (PAIR u u') subst env}

\quad \texttt{rewrite mergeCorrect u subst env | mergeCorrect u' subst env} \equiv \text{Refl}

\texttt{mergeCorrect (u \Rightarrow u') subst env}

\quad \texttt{rewrite mergeCorrect u subst env | mergeCorrect u' subst env} \equiv \text{Refl}

\texttt{notInMerge :: forall \{n m : Nat\} \rightarrow (i : \text{Fin n}) \rightarrow (\text{subst} : \text{Subst m n}) \rightarrow (\text{env} : \text{Env n}) \rightarrow (\text{env'} : \text{Env m}) \rightarrow \text{findInSubst} i \text{ subst} \equiv \text{Nothing} \rightarrow \text{findInEnv i \text{ env}} \equiv \text{findInEnv i (mergeClose subst env env')}

\texttt{notInMerge Fz (Cons Nothing \_)} (Cons \_ \_) \_ \_ \_ = \text{Refl}

\texttt{notInMerge Fz (Cons (Just \_ \_)) \_ \_ \_ (\_)}

\texttt{rewrite notInMerge i subst env env' eq}

\texttt{notInMerge (Fs i) (Cons Nothing subst) (Cons \_ \_ env) env' eq}

\texttt{rewrite notInMerge i subst env env' eq}

\texttt{inMerge :: forall \{n m : Nat\} \rightarrow (i : \text{Fin n}) \rightarrow (u : U m) \rightarrow (\text{subst} : \text{Subst m n}) \rightarrow (\text{env} : \text{Env n}) \rightarrow (\text{env'} : \text{Env m}) \rightarrow \text{findInSubst} i \text{ subst} \equiv \text{Just u} \rightarrow \text{elU u env'} \equiv \text{elU} (\text{VAR} i) (\text{mergeClose subst env env'})

\texttt{inMerge Fz (Cons Nothing \_)} (Cons \_ \_) \_ \_ \_ (\_)

\texttt{rewrite justInj eq | closing u env' eq}

\texttt{inMerge (Fs i) u (Cons Nothing subst) (Cons \_ \_ env) env' eq}

\texttt{rewrite inMerge i u subst env env' eq = Refl}

\texttt{inMerge (Fs i) u (Cons (Just \_ \_)) subst) (Cons \_ \_ env) env' eq}

\texttt{rewrite inMerge i u subst env env' eq = Refl}

\texttt{closing :: forall \{n : Nat\} \rightarrow (u : U n) \rightarrow (\text{env} : \text{Env n}) \rightarrow \text{elU} u \text{ env} \equiv \text{elU} (\text{close} u \text{ env})}

\texttt{closing NAT \_ \_ = \text{Refl}}

\texttt{closing (PAIR u u') \text{ env rewrite closing u env | closing u' \text{ env} = Refl}}

\texttt{closing (u \Rightarrow u') \text{ env rewrite closing u env | closing u' \text{ env} = Refl}}

\texttt{closing (VAR \_ \_) \_ = \text{Refl}}
Bibliography


Manuel Chakravarty. The Haskell 98 Foreign Function Interface 1.0, an addendum to the Haskell 98 report. 2003.


157


Summary

It is human nature to make errors, but we rather not see this reflected in software end-products. An established approach to detecting many of the errors in a program is to use a typed programming language where properties of operations and values are described by types. This allows a type system to verify that a program is well typed. Preferably, verification takes place statically during compilation such that errors are caught at an early stage and do not leak into the end-product. The disadvantage of static typing is that the type system is always an approximation and therefore sometimes wrongfully rejects programs.

The counterpart of static typing is dynamic typing, where verification is mostly in the hands of the programmer and takes place at run time by inspecting the types of values before coming to the actual operations. Consequently, any errors in a program will appear as late as when the program is executed. On the other hand, dynamic typing enlarges the number of programs that can be defined. The statically typed functional languages Clean and Haskell provide both static and dynamic typing. Clean offers dynamic typing via a built-in system whereas Haskell provides a less expressive library, but the underlying philosophy is the same: we start from a statically typed setting and provide a way out to perform dynamic typing. In essence, the use of dynamic typing is only really required when the types of values only become apparent at run time or when the static type system prevents us from defining a particular program. However, the static and dynamic type system are separate in Clean and Haskell. Consequently, whenever the language is extended, the effects on the static and dynamic type system have to be considered separately.

There is a clear trend visible in functional languages such as Clean and Haskell that types are not just used to verify a program, but also play another role. A more type-driven style of programming is emerging where the behaviour of a program also depends on the types of the values involved. Perhaps the earliest example is ad-hoc polymorphism via type classes where the behaviour of a function depends on the types occurring in the context where it is used. Another early form is generic programming where we define a function only once on a universe that describes the structure of types such that it can be applied to values of any type generically. Generalised algebraic datatypes enforce structural properties and even allow us to compute types from values. More recently, type families have taken the idea of type-driven programming to the next level by computing types from types. Dependent-typed languages such as Agda take this to an extreme since no real distinction is made between values and types: types are values.
Summary

All of the given examples of type-driven programming only concern static typing. However, dynamic typing is a form of type-driven programming as well. It involves making decisions based on the types of values, albeit at run time. This thesis describes in three parts how dynamic typing interacts with such static forms of type-driven programming.

The first part gives an overview of two functional languages, Clean and Haskell, that both play a leading role in this thesis. Chapter 2 describes an implementation of the Clean compiler to exchange sources between the two languages. Haskell programs can now take advantage of Clean’s more expressive dynamic type system.

The second part describes the necessity of dynamic typing in type-driven programming. Chapter 3 discusses a Haskell library that makes extensive use of type-driven programming via generalised algebraic datatypes and type families to generically rewrite terms using rewrite rules as values instead of functions. Dynamic typing turns out to be vital to define rewrite rules in terms of the original datatype instead of the inconvenient internal representations of the library. Chapter 4 focusses on manipulating heterogeneous structures described by generalised algebraic datatypes. Dynamic typing is needed to expose hidden type information, but is usually used in a cumbersome fashion. To make this more convenient, a new annotation is introduced and formally defined in terms of Clean’s dynamic typing.

The third part discusses how dynamic typing is accommodated in a language that facilitates type-driven programming. Chapter 5 focusses on ad-hoc polymorphism via type classes and explores the design space of the interaction with Clean’s dynamic typing. This brings a language where values seamlessly transfer between the static and dynamic world one step closer. Chapter 6 discusses what it takes for a functional language to embed polymorphic dynamic typing as a library, in contrast to Clean where this form of dynamic typing is built-in. The difficulties of such an embedding in Haskell are discussed and the need for a language such as Agda is motivated. Therewith, the parts are clarified where the additional power of a dependently-typed language is most needed, which gives great insight in what is required to perform such an embedding in Haskell.

This thesis shows that dynamic typing is an indispensable part of type-driven programming. It enriches the static forms of type-driven programming naturally since dynamic typing centers around types as well. However, it is not straightforward to accommodate dynamic typing in a language that facilitates type-driven programming, a built-in system such as in Clean seems to allow for better integration with the static forms of type-driven programming than a library.
Samenvatting

Fouten maken is menselijk, maar dat zien we liever niet terug in software-eind-producten. Een gevestigde aanpak voor het herkennen van veel van de fouten in een programma is het gebruiken van een getypeerde programmeertaal waarin eigenschappen van operaties en waarden zijn beschreven door types. Dit stelt een typeringssysteem in staat om te verifiëren dat een programma goed getypeerd is. Verificatie vindt bij voorkeur statisch plaats tijdens compilatie zodat fouten in een vroeg stadium worden herkend en niet in het eindproduct terecht komen. Het nadeel van statische typering is dat het typeringssysteem altijd een benadering is en daarom soms ten onrechte programma’s verwerpt.

De tegenhanger van statische typering is dynamische typering, waar verificatie voornamelijk in de handen van de programmeur ligt en plaats vindt tijdens de uitvoering van een programma door types van waarden te inspecteren alvorens tot de daadwerkelijke operaties te komen. Fouten in een programma zullen daarom pas tijdens de uitvoering van het programma aan het licht komen. Aan de andere kant vergroot dynamische typering het aantal programma’s dat kan worden gedefinieerd. De statisch getypeerde functionele talen Clean en Haskell bieden zowel statische als dynamische typering aan. Clean heeft dynamische typering ingebouwd terwijl Haskell een minder krachtige bibliotheek aanbiedt, maar de achterliggende filosofie is hetzelfde: we gaan uit van een statisch getypeerde omgeving en bieden een uitweg om dynamische typering uit te voeren. In wezen is het gebruik van dynamische typering alleen echt nodig wanneer de types van waarden pas bekend worden tijdens de uitvoering of wanneer het statische typeringssysteem ons weerhoudt van het definiëren van een bepaald programma. Het statische en dynamische typeringssysteem staan echter los van elkaar in Clean en Haskell. Dit heeft als gevolg dat bij een uitbreiding van de taal de effecten op het statische en dynamische typeringssysteem afzonderlijk moeten worden bekeken.

Er is een duidelijke trend zichtbaar in functionele talen zoals Clean en Haskell dat types niet alleen worden gebruikt om een programma te verifiëren, maar ook nog een andere rol spelen. Een meer typegestuurde stijl van programmeren is in opkomst waarbij het gedrag van een programma ook afhankt van de types van de betrokken waarden. Het vroegste voorbeeld is waarschijnlijk ad-hocpolymorfie via typeklassen waarbij het gedrag van een functie afhankt van de types in de context waarin het gebruikt wordt. Een andere vroege vorm is generiek programmeren waarbij een functie eenmalig wordt gedefinieerd op een universum dat de structuur van de types beschrijft zodat het generiek toepasbaar is op waarden van elk type. Gegeneraliseerde algebraïsche datatypes dwingen structurele eigenschap-
pen af en maken het zelfs mogelijk types vanuit waarden te berekenen. Het idee van typegestuurd programmeren is recentelijk naar het volgende niveau gebracht door typefamilies waarmee we types kunnen berekenen uit types. Zogenaamde dependently-typed talen zoals Agda gaan hier het verst in omdat er geen echt onderscheid wordt gemaakt tussen waarden en types: types zijn waarden.

Alle genoemde voorbeelden van typegestuurd programmeren betreffen alleen statische typering. Echter, dynamische typering is ook een vorm van typegestuurd programmeren. Het gaat om het nemen van beslissingen op basis van de types van waarden, zij het tijdens de uitvoering. Dit proefschrift beschrijft de wisselwerking tussen dynamische typering en zulke statische vormen van typegestuurd programmeren in drie delen.

Het eerste deel geeft een overzicht van twee functionele talen, Clean en Haskell, die beide een hoofdrol spelen in dit proefschrift. Hoofdstuk 2 beschrijft een implementatie van de Clean compiler om broncode tussen de twee talen uit te wisselen. Haskell programma’s kunnen nu gebruik maken van Clean’s krachtigere dynamische typeringssysteem.

Het tweede deel beschrijft de noodzaak van dynamische typering in typegestuurd programmeren. Hoofdstuk 3 bespreekt een Haskell bibliotheek die uitgebreid gebruik maakt van typegestuurd programmeren middels gegeneraliseerde algebraïsche datatypes en typefamilies om generiek termen te herschrijven en herschrijfregels als waarden in plaats van functies. Dynamische typering blijkt van levensbelang om herschrijfregels met het originele datatype te definieren in plaats van de onhandige interne representaties van de bibliotheek. Hoofdstuk 4 concentreert zich op het manipuleren van heterogene structuren beschreven middels gegeneraliseerde algebraïsche datatypes. Dynamische typering is nodig om verborgen type-informatie bloot te leggen, maar dit wordt meestal op een onhandige manier gedaan. Om dit te vergemakkelijken wordt er een nieuwe annotatie geïntroduceerd en formeel gedefinieerd in termen van Clean’s dynamische typering.

Het derde deel bespreekt hoe dynamische typering wordt ondergebracht in een taal die typegestuurd programmeren faciliteert. Hoofdstuk 5 concentreert zich op ad-hocpolymorfie via typeklassen en verkent de ontwerpruimte van de wisselwerking met dynamische typering in Clean. Dit brengt een taal waarin waarden naadloos kunnen worden overgedragen tussen de statische en dynamische wereld een stap dichterbij. Hoofdstuk 6 bespreekt wat een functionele taal nodig heeft om polymorfe dynamische typering met de taal zelf te definiëren in een bibliotheek, in tegenstelling tot Clean waar deze vorm van dynamische typering is ingebouwd. De moeilijkheid van zo’n definitie in Haskell wordt beschreven en de noodzaak van een taal zoals Agda wordt gemotiveerd. Daarmee wordt verduidelijkt waar de extra kracht van een dependently-typed taal het hardst nodig is, wat goede inzichten geeft in wat vereist is om het in Haskell zelf te definiëren.

Dit proefschrift laat zien dat dynamische typering een onmisbaar onderdeel is van typegestuurd programmeren. Het verrijkt de statische vormen van typegestuurd programmeren op een natuurlijke manier omdat dynamische typering ook om types draait. Het is echter niet eenvoudig om dynamische typering onder te brengen in een taal die typegestuurd programmeren faciliteert, inbouwen in de taal zelf lijkt een betere integratie met de statische vormen van typegestuurd programmeren te geven dan het onderbrengen in een bibliotheek.
Curriculum vitae

Thomas René van Noort

6 september 1984: geboren te Maarssen

september 1996 - juli 2002: VWO, Niftarlake College, Maarssen

september 2002 - juli 2005: BSc, Informatica, Universiteit Utrecht

september 2005 - januari 2008: MSc (cum laude), Informatica, Universiteit Utrecht

februari 2008 - januari 2012: AiO, ICIS, Radboud Universiteit Nijmegen