Abstract

Domain-specific languages (DSLs) are everywhere, with applications in areas such as parser generation, music synthesis, parallel programming and even the design of domain-specific languages. However, while the pay-off in using a DSL may be substantial, the cost of introducing a language may be made prohibitively high by the need to construct a supporting toolchain.

A common tactic is to embed a DSL into a general-purpose host programming language. Existing infrastructure such as a language’s compiler or type system may be re-used, provided that the embedding accurately captures the properties of the DSL. While the rich type systems and orthogonal abstraction features of modern functional languages have proved particularly capable in this regard, they are not without their shortcomings. Building type-safe functions defined over an embedded DSL can introduce application-specific type constraints that end up being imposed on the DSL data types themselves. At best, these constraints are unwieldy and at worst they can limit the range of DSL expressions that can be built.

In this thesis we tackle the problem of accurately embedding a DSL’s type system into that of the purely functional language Haskell. We present a framework for expressing application-specific constraints at the point of a DSL expression’s use rather than when the DSL’s embedding is defined. We show how our framework can be applied more generally to capture arbitrary properties of a DSL expression and, in certain cases, how we may subsequently prove additional safety properties such as the totality of a function which operates over DSL expressions. We evaluate our techniques by illustrating their use in constructing a DSL for heterogeneous parallel programming. However, our methods have potentially wider applications such as context-dependent computation, which are also discussed.
To my parents
I would like to thank my supervisor, Dr. Tony Field, for everything he has done for me throughout my time in the Department of Computing, from introducing me to Haskell in my first lecture as an undergraduate to tolerating my walking into his office unannounced almost every day for the last three years. While his knowledge, patience, encouragement and enthusiasm have undoubtedly contributed enormously to my producing this thesis, he is also solely responsible for my development as a teacher. I cannot imagine a better mentor and guide.

I owe much of my Haskell knowledge to the department’s functional programming group – Dr. Tristan Allwood. Be it the invaluable discussions in the corridor, the white board sessions in vacated offices, the arguments in the common room, the hours spent hacking and eating junk food or the uncountable number of ‘ah-hah!’ moments at my desk, my debts to Tristan are great. I truly couldn’t have done it without him.

To the SLURP reading group I owe the rest of my knowledge of language design, type theory and program verification. In particular I am extremely grateful to Professors Susan Eisenbach and Sophia Drossopoulou, who have always made time to listen to my numerous tirades, bouts of functional programming evangelism and practice talks and presentations.

I am grateful to my examiners, Dr. Conor McBride and Professor Sophia Drossopoulou, for making my viva such a pleasurable experience. Their extensive feedback and suggestions have no doubt improved the quality of this thesis significantly. Additionally, I would like to extend my thanks to Andres Löh and Ralf Hinze, whose work on lhs2TeX has made typesetting this tome of Haskell code possible.

Last but definitely not least, I’d like to thank my family and friends for their continuous support and understanding. While I dare not list names lest I forget someone, a special thanks must go to my parents, to whom this thesis is dedicated. After all, I wouldn’t be here without them.
## Contents

1 Introduction .................................................. 1
   1.1 Thesis structure and originality .......................... 3

2 Haskell's type system ......................................... 5
   2.1 Type classes and functional dependencies ................. 5
   2.2 Phantom types and empty types ............................ 9
   2.3 Equality constraints and GADTs .......................... 10
   2.4 Higher-rank types .......................................... 12
   2.5 Lexically-scoped type variables .......................... 15
   2.6 Type families and associated data types .................. 16
   2.7 Constraint kinds, rich kinds and kind polymorphism ...... 19

3 Embedding domain-specific languages ......................... 21
   3.1 Untyped and phantom-typed representations ................ 21
   3.2 GADTs and tagless encodings ............................. 24
   3.3 Polymorphic languages and parameterised type systems ..... 28
   3.4 Name binding and higher-order abstract syntax ............ 30
   3.5 Polyvariadic functions ...................................... 32
   3.5.1 Context sensitivity through type class overloading ..... 33
   3.5.2 Inferring types with global and local functional dependencies ... 35
   3.6 Type-level programming ..................................... 40
   3.6.1 Type families, type classes and functional dependencies .... 40
   3.6.2 Encoding properties with empty phantom types ........ 42
   3.7 Template metaprogramming and quasiquotation ............. 45
   3.8 Dependently-typed languages ............................... 47
   3.9 Discussion .................................................. 52

4 Deconstraining data types .................................... 53
   4.1 The type restriction problem revisited ...................... 53
   4.2 Generic constraints ......................................... 55
   4.2.1 Picking types ............................................. 57
   4.2.2 Platform-specific constraints ............................ 59
   4.2.3 Implementation cost ...................................... 61


List of Tables

2.1 A selection of types and their ranks. ........................................... 13

5.1 A subset of the antiquotation specifiers supported by Mainland’s C quasiquoting library ......................................................... 90
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Introducing a phantom type parameter, ( a ), to the Exp data type.</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Nikola [36] expression type definitions.</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Transforming the Exp type into a GADT.</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>The Idx and OpenExp GADTs of Accelerate [13]</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Unifying the BoolE and IntE constructors.</td>
<td>29</td>
</tr>
<tr>
<td>3.6</td>
<td>Using the types of the UnBOUND library [65].</td>
<td>31</td>
</tr>
<tr>
<td>3.7</td>
<td>Defining C’s printf function in Haskell using type class overloading.</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>Type inference in the presence of specialised class instances.</td>
<td>37</td>
</tr>
<tr>
<td>3.9</td>
<td>Type inference in the presence of local functional dependencies.</td>
<td>39</td>
</tr>
<tr>
<td>3.10</td>
<td>The Pack class of Kansas Lava.</td>
<td>41</td>
</tr>
<tr>
<td>3.11</td>
<td>The types of the HList library [33].</td>
<td>41</td>
</tr>
<tr>
<td>3.12</td>
<td>A selection of the Template Haskell types used for constructing Haskell ASTs.</td>
<td>46</td>
</tr>
<tr>
<td>4.1</td>
<td>The result of applying the dictionary-passing transformation [62] to the ((\in)) IntBool and AllIntBool type classes.</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>Combining the AllLeft and AllRight classes.</td>
<td>71</td>
</tr>
<tr>
<td>4.3</td>
<td>Encoding a Haskell type ( E ) and its ((\in)) constructors in Agda.</td>
<td>75</td>
</tr>
<tr>
<td>4.4</td>
<td>Encoding a class ( S ) of supported operations as a closed Agda data type.</td>
<td>77</td>
</tr>
<tr>
<td>4.5</td>
<td>Proving the totality of the compileCUDA function in Agda.</td>
<td>78</td>
</tr>
<tr>
<td>5.1</td>
<td>The TaglessExp source language.</td>
<td>82</td>
</tr>
<tr>
<td>5.2</td>
<td>Identity’s TaglessExp instance.</td>
<td>85</td>
</tr>
<tr>
<td>5.3</td>
<td>The saxpy function alongside the C code that might be generated on picking a suitable Floating/TaglessExp instance.</td>
<td>86</td>
</tr>
<tr>
<td>5.4</td>
<td>A TaglessExp instance for generating C code for unary, binary and ternary operator applications.</td>
<td>88</td>
</tr>
<tr>
<td>5.5</td>
<td>Inductively defining the compileCWith family of functions.</td>
<td>90</td>
</tr>
<tr>
<td>5.6</td>
<td>The C code generated as a result of compiling the saxpy function.</td>
<td>92</td>
</tr>
<tr>
<td>5.7</td>
<td>Wrapping scalar code generated by compileC with OpenMP parallel for-loops.</td>
<td>93</td>
</tr>
<tr>
<td>5.8</td>
<td>Using mapE’s rank-2 type to enable a task-parallel MappablePlatform instance.</td>
<td>95</td>
</tr>
<tr>
<td>5.9</td>
<td>Adapting the CCompilable class and its compileC method to produce the compileTH family of Template Haskell compilers.</td>
<td>98</td>
</tr>
</tbody>
</table>
Using type annotations to eliminate lists of types and their associated \((\in)\) constraints.

A function which exploits both heterogeneous parallelism through higher-rank types and specialisation through partial application.

Defining the instance \(\text{Num}\ (\text{Exp} \; \text{as} \; a)\) for implicitly polymorphic numeric literals and functions.

Rewriting \(\text{CCompilable}\) to handle functions over values of type \(\text{Exp}\).

Rewriting \(\text{CCompilable}\)'s recursive instance as an implementation of \(\text{lamE}\).

A fully overloaded implementation of the Black-Scholes formula for option pricing which may be lifted to operate over values of type \(\text{Exp}\) using \(\text{polyLamE}\).

Using the \(\text{Rep}\) type family to type the application of \(\text{lamE}\) to the \(\text{swap}\) function.

Deriving generic constraints for Chapter 4's \(\text{Exp}\) GADT using Template Haskell.

Constraining context-dependent functions using generic constraints.
Domain-specific languages (DSLs) have become an indispensable tool in the development and specification of software, offering high-level abstractions while removing the need to write extraneous 'boilerplate' code. Embedding a DSL into a general-purpose programming language in the manner of Leijen and Meijer [34] or Augustsson et al. [5], for example, provides a simple and effective way to support domain-specific functionality without the need for a custom toolchain. Modern functional languages have proved to be particularly powerful hosts for DSLs due in part to their rich type systems which, appropriately exploited, can endow the DSL with important static safety properties. For example, one may wish to ensure that a database query is well-typed with respect to a given schema, or validate that a given railway design does not connect tracks with opposing directions of travel.

In this thesis we will focus on DSLs which facilitate code generation for heterogeneous parallel computing platforms. The aim is to expose a domain-specific subset of the host language in which parallel programs may be both written and executed on a target platform. Importantly, we are concerned with solving a variant of Wadler’s expression problem [61], in which the goal is to write a parallel program without knowledge of the target platform whilst also being able to introduce new platforms without breaking existing code. Moreover, whether or not a program and target platform are compatible should be statically decidable and should not require the program to be run. Examples of the kinds of incompatibilities we want to avoid statically include:

- An FPGA capable only of fixed-point integer arithmetic should be prohibited from running an application that uses floating-point types or operations.
- A GPGPU’s SIMD processors should be prohibited from running divergent programs, such as those which make use of higher-order functions or recursion.
- Optimising a program using SSE vectorisation should be prohibited if the program uses values which aren’t single-precision floating-point numbers.

The difficulty in providing such safety guarantees is that they must be expressed in terms of the existing features of the host language. Depending on the similarity of the DSL and its host, this may result in an embedding in which invalid programs can be written or, worse, in which valid programs are deemed invalid. In the purely functional language Haskell (Peyton Jones et al. [48]), the latter scenario typically arises as a result of overly-strict type class constraints (Peyton
Jones et al. [45]) which, while preventing the expression of ill-typed or incompatible DSL programs, do exactly what they say: they constrain the way the DSL can be used. This problem is amplified when a DSL is to support multiple implementations (possibly simultaneously), each of which possesses its own set of constraints. For example, consider the following code, which exemplifies the sort of programs we want to write:

\[
\begin{align*}
  f \ x \ y &= \text{if } y \equiv 0 \text{ then } x \text{ else } x / y \\
  zs &= \text{using (cpu || gpgpu) runStream (zipWithE \ f \ xs \ ys)}
\end{align*}
\]

We want to be sure that our program is well-typed if and only if \( f \) is compatible with the cpu and gpgpu platforms. If, for example, the stream \( xs \) consists of values of a type supported by the CPU but not the GPGPU, we should expect a static type error. Moreover, we might also desire a similar error in the event that, for instance, the division operation used by \( f \) is not supported by either platform. To enable the type checker to produce such errors, we must furnish it with type-level descriptions of both the requirements of \( f \) and the capabilities of the composition of the cpu and gpgpu platforms, as well as a mechanism for testing their compatibility. What makes this difficult is that \( f \) has no idea which platforms it will target, meaning that we cannot constrain the function at the point of its definition. Similarly, baking knowledge of \( f \) into the definitions of the cpu and gpgpu platforms would cripple their usefulness, and precludes one from freely introducing new platforms. The challenge therefore is to impose the necessary constraints retrospectively, at the point at which a program is to be executed on a chosen platform.

In this thesis we explore how heterogeneous DSLs may be embedded in Haskell in a type-safe manner. In particular, we are interested in embedding a subset of Haskell within itself, the idea being that existing polymorphic Haskell programs may be used without modification wherever possible. While features such as operator overloading, pattern-matching and type inference are key to writing idiomatic Haskell programs, we illustrate how their presence in a heterogeneous embedded DSL can make enforcing typesafety difficult. We develop techniques for overcoming these difficulties and evaluate them not only with respect to the safety guarantees that they afford, but also by considering the impact they have on the ease with which a DSL utilising such solutions may be embedded and used.

Our contributions are as follows:

**Deconstraining data types**

We describe a method for imposing generic constraints on data types that avoids the restrictions ordinarily imposed by concrete constraints (Chapter 4, based in part on the paper 'Deconstraining DSLs' [28]). In the context of embedding a DSL we show how these constraints can be used to impose independent implementation-specific constraints, for example on both the types (Section 4.2.2) and operations (Section 4.5) supported by a given target platform. While our constraints are initially presented using generalised
algebraic data types, we see that they can be applied equally successfully to tagless representations (Section 4.4), even in the presence of higher-rank types.

We show that both the compile-time and run-time overheads of the scheme are bounded by a constant that is a function of the size of the list of types forming the constraint set, which we expect typically to be small (Section 4.2.3). Furthermore, we provide a translation from programs which use our technique to constrain operations to provably-total Agda (Norell [42]) programs (Section 4.6), which we believe validates the correctness of our methodology.

A type-safe heterogeneous language

We conduct a case study (Chapter 5) in which we develop a single DSL which resembles ordinary Haskell code, showing in the process how generic constraints do not impede our ability to use Haskell’s support for operator overloading (Section 5.1). Aside from targeting Haskell itself, we show how back ends can be written which generate C code at run-time (Section 5.3) and Template Haskell (Sheard and Peyton Jones [53]) code at compile-time (Section 5.4). Finally, we demonstrate the value of our technique’s main departure from related work—its compatibility with higher-rank types—by showing how back ends may be combined to construct heterogeneously-parallel code (Section 5.3.2) from a single DSL expression or function instantiated multiple times at different types.

Higher-order embeddings and the boundaries of Haskell’s type system

We consider how we may further reduce the gap between Haskell’s type system and that of a DSL to be embedded (Chapter 6) by providing type-level constructs for generically constraining function types (Section 6.3) and permitting the use of pattern-matching over algebraic data types (Section 6.5.1). In doing so we test the limits of many powerful extensions to Haskell’s type system, including type class overloading (Peyton Jones et al. [45]), kind polymorphism (Yorgey et al. [66]), constraint kinds, data type promotion, indexed type families (Schrijvers et al. [51], Chakravarty et al. [12]) and higher-rank types (Vytiniotis et al. [60], Peyton Jones et al. [46]). We provide insight into how these features interact with each other and reflect on the limitations (Section 6.6) and difficulties (Section 5.5) that arise as a result of their combined use.

1.1 Thesis structure and originality

Chapters 2 and 3 give an overview of Haskell’s type system and its applications in embedding domain-specific languages. Chapter 4 is based on the paper ‘Deconstraining DSLs’ [28] and is thus a fairly self-contained starting point for readers well versed in such background material. Chapters 5 and 6 comprise new work which applies and extends the techniques presented in the paper. Much of this thesis has been proof-read several times by my supervisor Dr. Tony Field, whose detailed suggestions have been incorporated throughout. Furthermore, I, Dr. Field
and Dr. Tristan Allwood co-authored the paper 'Deconstraining DSLs' [28], on which parts of this introduction, Chapter 4 and Chapter 7 are based. Except where otherwise referenced, all remaining contributions are my own.
Throughout this thesis we shall make use of many of the more powerful aspects of Haskell's type system, including generalised algebraic data types (GADTs), higher-rank types, type families and rich kinds. For the sake of providing a self-contained narrative we will give a brief overview of these features here. The reader already familiar with these concepts may therefore wish to skip directly to Chapter 3, in which we discuss the applications of Haskell's type system in embedding DSLs, or Chapter 4, which contains material from the paper 'Deconstraining DSLs' [28] and is thus fairly self-contained in its own right.

We remark that many of the features used in this thesis (both at the type and value level) are provided as extensions to the Glasgow Haskell Compiler (GHC) and are thus not strictly part of Haskell as defined in, for example, Haskell 98 (Peyton Jones et al. [48]). Despite this we will commonly use terms such as 'Haskell's type system' to refer to, for example, the Haskell 98 type system as extended by GHC. Familiarity with Haskell's basic features such as ad-hoc polymorphism, abstract data types and pattern matching is assumed throughout this thesis.

2.1 Type classes and functional dependencies

Functions in Haskell may be overloaded through the use of type classes, as described by Wadler and Blott [62]. The Eq class, for instance, captures the set of types whose values may be compared under equality:

```haskell
class Eq a where
  (≡), (\neq) :: a → a → Bool
```

The (≡) and (\neq) methods may be used to compare values of any type which is an instance of the Eq class. For example, the Bool type itself possesses the following instance:

```haskell
instance Eq Bool where
  True ≡ True = True
  False ≡ False = True
  _  ≡ _  = False
```

A definition for (\neq) is safely omitted due to the fact that the Eq class provides a default implementation in terms of (≡):
class Eq a where

\[ x \not\equiv y = \text{not} (x \equiv y) \]

Type classes may specify \textit{superclass constraints}, in which a type may only be made an instance of a class if the appropriate superclass constraints are met. The \textit{Ord} class, for example, represents those types whose values may be ordered, provided that they are also comparable under equality:

\[
\text{class } \text{Eq } a \Rightarrow \text{Ord } a \text{ where} \\
(\langle), (\rangle), (\leq), (\geq) :: a \rightarrow a \rightarrow \text{Bool} \\
\text{min, max} :: a \rightarrow a \rightarrow a
\]

Here, the constraints listed before the \( \Rightarrow \) symbol (Eq \( a \) in this case) are referred to as the \textit{context}, whilst the declaration following it is known as the \textit{head}. These definitions also apply in the case of instances; lists of values, for example, may only be compared under equality if the elements are themselves comparable:

\[
\text{instance Eq } a \Rightarrow \text{Eq } [a] \text{ where} \\
[] \equiv [] = \text{True} \\
(x : xs) \equiv (y : ys) = x \equiv y \land xs \equiv ys \\
\_ \_ \equiv \_ \_ = \text{False}
\]

where Eq \( a \) is the instance context and Eq \([a]\) the instance head. There is an important different between these two uses, however. In the definition of Ord, the constraint Eq \( a \) encompasses all possible instances of Ord \( a \): whenever the context Ord \( a \) is known, the context Eq \( a \) will be also. In the latter case, the context Eq \( a \) is available to the Eq \([a]\) instance only; the Eq \([a]\) instance may only be selected if the constraint Eq \( a \) is satisfiable.

Haskell 98 imposes several restrictions on the definition and use of type classes. Most notable is that type classes must be parameterised over exactly one type. GHC lifts this restriction and supports \textit{multi-parameter type classes}, allowing such classes as:

\[
\text{class \ Collects } c \ a \text{ where} \\
\text{empty } :: c \\
\text{insert } :: a \rightarrow c \rightarrow c
\]

Collects is a \textit{relation} between pairs of types \( c \) and \( a \). In the example, \( c \) is intuitively a collection containing elements of type \( a \). For example, an instance for working with Haskell’s lists might look thus:

\[
\text{instance Collects } [a] \ a \text{ where} \\
\text{empty } = [] \\
\text{insert } = (:)
\]
There is a flaw, however: it is impossible to use the empty function. Looking at its type reveals why:

\[
\text{empty :: Collects } c \ a \Rightarrow c
\]

The type variable \(a\) is ambiguous – the compiler is unable to infer what \(a\) might be and consequentially cannot decide which overloading of the empty function to pick. The problem is that Collects is a relation over types, when what is really needed is a partial function. Functional dependencies as described by Jones [27] are another GHC extension suited to just this task. The Collects type class can be refined to become:

```haskell
class Collects c a | c \rightarrow a where
    empty :: c
    insert :: a \rightarrow c \rightarrow c
```

where the dependency \(c \rightarrow a\) states that for any type \(c\), there can be at most one type \(a\) associated with it through the Collects class. More succinctly, \(c\) determines \(a\). Returning to the type of empty:

\[
\text{empty :: Collects } c \ a \Rightarrow c
\]

As before, calling empty requires knowledge of the type \(c\). However, since \(c\) determines \(a\), the compiler may now infer the type \(a\) given \(c\); the type is thus unambiguous and the function usable.

A remaining caveat is that the earlier instance Collects \([a]\) \(a\) is not valid Haskell 98. In Haskell 98, instance heads must constitute an application \(C (T a_1 \ldots a_n)\), where \(C\) is a type class, \(T\) is a type constructor and \(a_1 \ldots a_n\) are type variables. Similarly, contexts must be of the form \(C a_1 \ldots a_n\), where \(C\) is a type class and \(a_1 \ldots a_n\) are type variables. GHC provides several mechanisms for relaxing these rules:

**Flexible instances**

The head of an instance declaration may mention arbitrary type applications, e.g.:

```haskell
instance C [Int] where
    ...
```

**Overlapping instances**

Instances may overlap, provided that there is a most specific one. That is to say, given a context \(C [Int] \ Int\) and the instances:

```haskell
instance C a \ Int
instance C Bool a
```
instance \( C \ [a] \) \ Int
instance \( C \ [Int] \) \ Int

the compiler will select the last instance, which is a perfect match, despite the fact that the first and third are also possible candidates. When selecting an instance, GHC examines only the head – the context is introduced as a constraint once an instance has been matched.

**Flexible contexts**

Contexts need not be of the form \( C \ a_1 \ldots a_n \) provided that they satisfy the following conditions:

- The Paterson conditions, proposed by Ross Paterson: for every constraint in a context, no type variable occurs more frequently than it does in the head. Furthermore, each constraint must have fewer constructors and variables than the head.

- The coverage condition: given a substitution \( S \) which maps each type variable in a class declaration to a corresponding type in an instance declaration, it must be the case that each of the class' functional dependencies is of the form \( as \to bs \), where \( S(bs) \subset S(as) \).

**Undecidable instances**

As Sulzmann et al. [55] show, the Paterson and coverage conditions are sufficient to guarantee that the process of context reduction—solving the constraints imposed on a program by class contexts—remains decidable. Unfortunately there are times where they reject provably terminating instances; for example:

```haskell
class \( C \ a \) where
  ... 
class \( D \ a \) where
  ... 
instance \( C \ a \) where
  ... 
instance \( C \ a \Rightarrow D \ a \) where
  ... 
```

Here the second instance bears a constraint which is no smaller than the head, violating the Paterson conditions. Only considering one instance at a time, the type checker will thus reject the program based on the fact that an instance such as \( D \ a \Rightarrow C \ a \), say, would prevent context reduction from termination. However, if we consider both instances simultaneously, we see that, in the absence of overlapping instances, type checking will always terminate – there can be no other instances of \( C \) which may cause the type
checker to loop. Since termination is only enforced on a per-instance basis, however, the program is rejected. GHC may be instructed to lift these final restrictions, permitting the programmer to write such instances. Of course, non-terminating instances may now also be written; it is thus an instruction to be issued with care.

2.2 Phantom types and empty types

Consider the following definitions:

```haskell
data T a b = T a
  t1 :: T Int Bool
  t1 = T 0
  t2 :: T Int Int
  t2 = T 0
```

Here `t1` and `t2` will have identical run-time representations; the type variable `b` in `T`'s definition (here instantiated to `Bool` and `Int` in `t1` and `t2` respectively) exists only in the eyes of the type checker. We say that `b` (or sometimes even `T`) is a *phantom type* for this reason.

Phantom types are not so much a feature of Haskell's/GHC's type system but a tool with which other features may be combined. *Empty data types*, as supported by GHC, are particularly useful in this instance. The type-level natural numbers, for example, may be defined as follows:

```haskell
data Zero

data Suc n
```

Here the types `Zero` and `Suc n` are empty; no values of these types exist apart from `⊥`. This mirrors nicely the purpose of a phantom type, which encodes type-level information without impacting a program's run-time representation. For instance, we may define:

```haskell
newtype Vec a n = Vec [a]
```

Here, it is intended that `Vec a n` be the type of lists of elements of type `a` whose length is statically known to be `n`. Regrettably this property cannot be enforced by the `Vec` constructor, which possesses the type:

`Vec :: [a] → Vec a n`

Here the length `n` is free and so it is easy to create vectors which lie about their lengths:

`Vec []` :: Vec Int (Suc Zero)
`Vec [True, False]` :: Vec Bool Zero
Consequently we must opt not to make the Vec constructor visible, instead defining a pair of smart constructors, nilV and consV, which restrict the actual constructor’s type appropriately:

\[
\begin{align*}
nilV :: Vec \ a \ Zero \\
nilV &= Vec \ [] \\
consV :: a \rightarrow Vec \ a \ n \rightarrow Vec \ a \ (Suc \ n) \\
consV \ x \ (Vec \ xs) &= Vec \ (x : xs)
\end{align*}
\]

Assuming that these are now the only functions which may be used to build Vecs, we have obtained a level of type safety that wasn’t previously available. Consider:

\[
\begin{align*}
headV :: Vec \ a \ (Suc \ n) \rightarrow a \\
headV (Vec \ (x : xs)) &= x
\end{align*}
\]

A Vec with size Suc \ n (for some \ n) must have been built by consV, so headV is ‘total’ in practice. Of course, the compiler cannot verify this for itself and this property will vanish if the Vec constructor is ever exported. Despite this, phantom types have seen use in many applications, examples including foreign function interfaces (Finne et al. [21]), subtyping (Fluet and Pucella [22]) and, of course, embedded domain-specific languages (Leijen and Meijer [34]).

### 2.3 Equality constraints and GADTs

The previous section details an approach to constructing length-annotated Vecs that relies on a well-designed interface to maintain type safety, due entirely to the fact that the Vec constructor’s type is too liberal. *Generalised algebraic data types* (GADTs) as provided by GHC (Cheney and Hinze [14], Schrijvers et al. [52]) allow one to solve this problem by specifying the types of a data type’s constructors explicitly. The Vec type, for example, may be recast as:

\[
\textbf{data} \ Vec \ a \ n \ \textbf{where} \\
\quad \text{NilV} :: Vec \ a \ Zero \\
\quad \text{ConsV} :: a \rightarrow Vec \ a \ n \rightarrow Vec \ a \ (Suc \ n)
\]

Observe that the difference between an ADT and a GADT is that a GADT may introduce both existential quantification and equality constraints, which restrict a constructor’s type. In our case:

\[
\begin{align*}
\text{NilV} :: (n \sim Zero) \Rightarrow Vec \ a \ n \\
\text{ConsV} :: (n \sim Suc \ m) \Rightarrow a \rightarrow Vec \ a \ m \rightarrow Vec \ a \ n
\end{align*}
\]

where the constraint \((n \sim Zero)\) states that \(n\) is equal to \(Zero\) and the type variable \(m\) does not appear in the visible result type of ConsV. Equality constraints facilitate type refinement.
through pattern-matching: in the rewritten headV function, for example, attempting to add a case headV NilV = ... is a type error. Pattern-matching the NilV constructor introduces the knowledge that the argument is of type Vec a Zero, when it is required to have type Vec a (Suc n). The function is thus total, and recognised as such by the compiler:

\[
\begin{align*}
\text{headV} :: & \text{Vec } a \ (\text{Suc } n) \to a \\
\text{headV} (\text{Cons} \ x \ xs) &= x
\end{align*}
\]

Cheney and Hinze \[14\] refer to GADTs as 'first-class phantom types'. Where a (second-class) phantom type allows one to write a smart constructor only, a GADT permits one to also write a smart destructor. For example, given a small expression language:

```haskell
data Exp a where
  BoolE :: Bool \to Exp Bool
  IntE :: Int \to Exp Int
  AddE :: Exp Int \to Exp Int \to Exp Int
  EqE :: Exp Int \to Exp Int \to Exp Bool
```

a type-safe evaluator may be written thus:

```haskell
evaluate :: Exp a \to a
evaluate (BoolE x) = x
evaluate (IntE x) = x
evaluate (AddE e1 e2) = evaluate e1 + evaluate e2
evaluate (EqE e1 e2) = evaluate e1 \equiv evaluate e2
```

Contrast this with an implementation involving an ordinary ADT, in which this would not be possible:

```haskell
data Exp a = BoolE Bool | ...
```

```haskell
evaluate :: Exp a \to a
evaluate (BoolE x) = x
```

The type of BoolE is now Bool \to Exp a. The above clause thus has type Exp a \to Bool, which the compiler cannot unify with the function’s declared type Exp a \to a. The evaluate function is thus ill-typed.

GADT constructors may in fact be constrained by arbitrary contexts. It is possible to construct a list of values which may be compared under equality, for example, as follows:
\textbf{data} EqList \ a \ \textbf{where}
\begin{align*}
\text{NilL} & : \text{EqList} \ a \\
\text{ConsL} & : \text{Eq} \ a \Rightarrow a \rightarrow \text{EqList} \ a \rightarrow \text{EqList} \ a
\end{align*}

\textbf{instance} Eq (EqList \ a) \ \textbf{where}
\begin{align*}
\text{NilL} & \equiv \text{NilL} \quad = \text{True} \\
\text{ConsL} \ x \ xs & \equiv \text{ConsL} \ y \ ys \equiv x \equiv y \wedge xs \equiv ys \\
_ & \equiv _ \quad = \text{False}
\end{align*}

Note that unlike the \textit{Eq} instance for \([a]\), there is no need for a superclass constraint such as \textit{Eq} \ \ a – pattern-matching the \textit{ConsL} constructor introduces exactly that context. Embedding context in a constructor's type is especially useful in the presence of \textit{existential types}. The \textit{Exp} type presented above, for instance, could be generalised as follows:

\textbf{data} Exp \ a \ \textbf{where}
\begin{itemize}
\item ...
\item EqE :\ Eq \ a \Rightarrow \text{Exp} \ a \rightarrow \text{Exp} \ a \rightarrow \text{Exp} \ \text{Bool}
\end{itemize}

Here, the type variable \(a\) in \textit{EqE}'s signature is existential: it is hidden from consumers of the \textit{Exp} type. Consequently, pattern-matching an \textit{EqE} constructor reveals nothing about the values it contains other than the fact that they may be compared with each other under equality. This is all we need for the \textit{evaluate} function to remain correct:

\begin{align*}
\text{evaluate} & : \text{Exp} \ a \rightarrow a \\
\text{evaluate} \ (\text{EqE} \ e_1 \ e_2) & = \text{evaluate} \ e_1 \equiv \text{evaluate} \ e_2
\end{align*}

Without a context such as \textit{Eq} \ \ a, the compiler would only know that the arguments of an \textit{EqE} constructor application have the \textit{same} unknown type.

\subsection{2.4 Higher-rank types}

In Haskell 98, all type variables are implicitly universally quantified. For example, the type of the \textit{const} function, given by:

\begin{align*}
\text{const} & : a \rightarrow b \rightarrow a \\
\text{const} \ x \ y & = x
\end{align*}

can be made progressively more explicit as follows:

\begin{align*}
\text{const} & : a \rightarrow b \rightarrow a \\
\text{const} & : a \rightarrow (b \rightarrow a)
\end{align*}
<table>
<thead>
<tr>
<th>Type</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int → Bool → Int</td>
<td>0</td>
</tr>
<tr>
<td>∀a b. a → b → a</td>
<td>1</td>
</tr>
<tr>
<td>Int → (∀a b. a → b → a)</td>
<td>1</td>
</tr>
<tr>
<td>∀a. a → (∀b c. b → c → b) → a</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2.1: A selection of types and their ranks.

const :: ∀a. a → (∀b. b → a)
const :: ∀a b. a → (b → a)

The rank of a type can be thought of as a measure of how much quantification it introduces. Kfoury and Wells [32] define the rank of a type as follows: let \( R(0) \) be the set of open types, that is types which do not mention the symbol ‘∀’ (i.e. introduce any quantification). \( R(k + 1) \) is then given by the grammar:

\[
R(k + 1) \::= R(k) \\
| (R(k) \rightarrow R(k + 1)) \\
| (∀V. \ R(k + 1))
\]

where \( V \) is the set of type variables. Table 2.1 gives some examples of types and their ranks according to this definition. Type inference in the presence of higher-rank types is in general undecidable (Kfoury and Tiuryn [31], Kfoury and Wells [32]). Definitions wishing to make use of such types must therefore be annotated by the programmer.

A particularly important use-case of higher-rank types is Wadler and Blott’s dictionary-passing transformation [62] of type classes, utilised by GHC and many other Haskell compilers to implement type classes internally. Consider once more the Eq class, along with a function that requires an Eq context:

```haskell
class Eq a where
    (≡), (≠) :: a → a → Bool
allEqual :: Eq a ⇒ a → a → a → Bool
    allEqual x y z
    = x ≡ y ∧ y ≡ z
```

The transformation realises the Eq class as a data structure, termed a dictionary:

```haskell
data Eq a
    = Eq (≡), (≠) :: a → a → Bool
```

The dictionary is a record with fields corresponding to each method of the class it embodies, here the (≡) and (≠) functions. Contexts of the form Eq a now become additional function arguments, while class methods are rewritten as projections over those arguments:
allEqual :: Eq a → a → a → a → Bool
allEqual d x y z = (≡) d x y ∧ (≡) d y z

In the context of dictionary-passing transformations, higher-rank types are necessary when type class methods (even implicitly) introduce universal quantification over variables by which the class is not parameterised. One example is the Functor class, which generalises the map operation over lists:

```haskell
class Functor f where
  fmap :: (a → b) → f a → f b
```

The class’ dictionary looks as follows:

```haskell
data Functor f
    = Functor { fmap :: ∀a b. (a → b) → f a → f b }
```

Note that the Functor dictionary constructor possesses a rank-2 type:

```haskell
Functor :: ∀f. (∀a b. (a → b) → f a → f b) → Functor f
```

As an example of what this polymorphism offers, the following Haskell predicate checks the second functor law for a given functor, \( f \), and point \( x \):

```haskell
p :: (Eq (f c), Functor f) ⇒ (b → c) → (a → b) → f a → Bool
p f g x = fmap (f · g) x ≡ (fmap f · fmap g) x
```

Applying the dictionary-passing transformation to this function yields a definition with arguments for the Eq \( (f c) \) and Functor \( f \) constraints as follows:

```haskell
p :: Eq (f c) → Functor f → (b → c) → (a → b) → f a → Bool
p de df f g x = (≡) de (fmap df (f · g) x)
  ((fmap df f · fmap df g) x)
```

Here, fmap \( df \) is being applied three times at three different types:

- fmap \( df (f · g) :: f a → f c \)
- fmap \( df f :: f b → f c \)
- fmap \( df g :: f a → f b \)

If the Functor dictionary’s constructor did not possess a rank-2 type, \( p \) would not type check. Assuming that the compiler reaches the application fmap \( df (f · g) \) first, for example, fmap \( df \)'s type will be instantiated with the variables \( a \) and \( c \) first, for example, fmap df’s type which, being completely specified by the type signature of \( p \), are said to be rigid – they are assumed distinct. The required unification of either \( a \) and \( b \) or \( c \) and \( b \) prompted by the remaining uses of fmap df will thus fail.
2.5 Lexically-scoped type variables

Consider the following program:

\[
\begin{align*}
  f &:: [a] \to [a] \\
  f \; xs &= xs + ys \\
  \text{where} & \\
  ys &= \text{reverse} \; xs
\end{align*}
\]

In Haskell 98 it is impossible to give the subexpression \(ys\) a type signature. While we might wish to introduce one, viz.:

\[
\begin{align*}
  f &:: [a] \to [a] \\
  f \; xs &= xs + ys \\
  \text{where} & \\
  ys &:: [a] \\
  ys &= \text{reverse} \; xs
\end{align*}
\]

we cannot: the implicit quantification of type variables mentioned in the previous section means that the above program actually results in two distinct type variables \(a\):

\[
\begin{align*}
  f &:: \forall a. \; [a] \to [a] \\
  f \; xs &= xs + ys \\
  \text{where} & \\
  ys &:: \forall a. \; [a] \\
  ys &= \text{reverse} \; xs
\end{align*}
\]

Clearly, \(ys\) cannot have the type \(\forall a. \; [a]\) as it depends on the value of \(xs\) (whose type is fixed at the point of applying \(f\)) – indeed, the only value with the type \(\forall a. \; [a]\) is the empty list (\([\,]\)). The program is thus type incorrect.

GHC supports lexically-scoped type variables, whereby the programmer is afforded some control over whether the use of a type variable introduces quantification or references an existing variable already in scope. For example, we are permitted to rewrite the program above as follows:

\[
\begin{align*}
  f &:: \forall a. \; [a] \to [a] \\
  f \; xs &= xs + ys \\
  \text{where} & \\
  ys &:: [a] \\
  ys &= \text{reverse} \; xs
\end{align*}
\]

Now the variable \(a\) in \(f\)'s type signature is quantified by virtue of the explicit syntax \(\forall a.\), while the variable \(a\) in \(ys\)'s type signature is not so quantified and so references the same type variable.
A variable need not be bound by the type signature of a declaration or expression; the function \( f \) could also be recast using a pattern type signature as follows:

\[
f :: [a] \rightarrow [a] \\
f (xs :: [b]) = xs ++ ys \\
\textbf{where} \\
y :: [b] \\
y = \text{reverse} \, xs
\]

Here the value \( xs \) is annotated with the type \([b]\), bringing the type variable \( b \) into scope and making it possible to type \( ys \) once more. It is not necessary to use a quantifier here, as in \( f (xs :: \forall b. \, [b]) \). Indeed, this would state that \( xs \) is a polymorphic argument and that \( f \) possesses a higher-rank type.

Note also that we have already been exposed to lexically-scoped type variables in the form of type class and instance declarations – recall, for example, the \( \textit{Functor} \) type class of Section 2.4:

\begin{verbatim}
class Functor f where 
  fmap :: (a -> b) -> f a -> f b
\end{verbatim}

Here the type variable \( f \) scopes over the type of \( \text{fmap} \), which only quantifies over the type variables \( a \) and \( b \).

### 2.6 Type families and associated data types

Traditional Haskell type constructors are \textit{parametric}, defining a type whose representation is independent of the types it is applied to. The list type illustrates this as well as any other:

\begin{verbatim}
data [a] = [] | a : [a]
\end{verbatim}

All lists are either empty ([]) or non-empty (x:xs), irrespective of the element type. In contrast, GHC’s \textit{indexed type families} (or often just type families) (Chakravarty et al. [12], Sulzmann et al. [54], Schrijvers et al. [51]) provide the ability to define type constructors which specialise their representation depending on supplied type parameters. One example of where this is extremely useful is \textit{data-parallel programming}. Accelerate (Chakravarty et al. [13]), REPA (Keller et al. [30]) and data-parallel Haskell (DPH) (Peyton Jones et al. [47]) are all examples of data-parallel programming libraries which make use of indexed type families in order to efficiently represent data. DPH, for example, introduces the type \([:: a ::]\) of ‘parallel arrays’ of values of type \( a \):

\begin{verbatim}
data family [:: a ::]
\end{verbatim}

While similar in appearance to the definition of the list type above, the \textbf{family} keyword in this definition introduces a family of types, \([:: ::]\), \textit{indexed (not parameterised)} by a type \( a \). Here
the idea is that an array’s representation depends on the type of element it is to hold. An array of single-precision floating-point numbers, for example, might be stored as a tightly packed, unboxed sequence; this can defined as an instance of the \([\cdot \cdot\cdot]\) family:

```haskell
data instance :\(\cdot\) Float :\]
\quad \Rightarrow \text{FloatPA} (\text{UnboxedSequence Float})
```

Similarly, an array of pairs would be better represented as a pair of arrays:

```haskell
data instance :\((a, b)\) :\]
\quad \Rightarrow \text{PairPA} [:a:] [:b:]
```

In this way each type may be associated with a representation amenable to parallel execution. In doing so though, it appears as though we have lost the ability to write representation-agnostic functions such as:

```haskell
\text{lengthP} :: [:a:] \rightarrow \text{Int}
```

Fortunately, because function overloading is already supported by type classes, we can recover this facility by pairing all instances [:a:] with an appropriate instance of a suitable type class:

```haskell
\text{class} \text{PAElem} a \text{ where}
\quad \text{lengthP} :: [:a:] \rightarrow \text{Int}

\text{instance} \text{PAElem} \text{Float} \text{ where}
\quad \text{lengthP}
\quad \quad = \ldots

\text{instance} \ (\text{PAElem} \ a, \text{PAElem} \ b) \Rightarrow \text{PAElem} \ (a, b) \text{ where}
\quad \text{lengthP}
\quad \quad = \ldots
```

This is fine, but it would be more natural to somehow couple the family [:\cdot\cdot\cdot:\] with the class PAElem. Thankfully, GHC’s associated type families do just that:

```haskell
\text{class} \text{PAElem} a \text{ where}
\quad \text{data} [:a:]
\quad \text{lengthP} :: [:a:] \rightarrow \text{Int}
```

This definition states that types which are instances of the PAElem class must declare also an instance of the [:a:] data type family. The family keyword is omitted in this context since the presence of a family of types is implicit. Instances follow as one might expect (here, lengthUS returns the length of an UnboxedSequence):

```haskell
\text{instance} \text{PAElem} \text{Float} \text{ where}
\quad \text{data} [:\text{Float}:]
```
Strictly speaking, the family of types \([\_ / a \_]\) is a \textit{data type family} (or just data family); it introduces constructors in the same manner as a vanilla Haskell data type. \textit{Type synonym families} (often confusingly referred to as type families themselves) are the equivalent analogue of Haskell’s type synonyms, which introduce new names for existing types:

\begin{verbatim}
  type family Element a
  type instance Element [a] = a
  type instance Element (Maybe a) = a
\end{verbatim}

Here the Element family intuitively represents the type-level function from a container of values to the type of its elements; it thus has parallels with the \(a\) parameter of the \texttt{Collects} class of Section 2.1. Indeed, type families often present a more functional alternative to the logical mode of expression provided by functional dependencies, particularly when associated with a type class.

Of note is that synonym families are not \textit{injective}, that is to say that for a family \(\mathcal{F}\), the constraint \(\mathcal{F} a \sim \mathcal{F} b\) does not imply that \(a \sim b\). For example, the following definitions are perfectly legal:

\begin{verbatim}
  type family \(\mathcal{F}\) a
  type instance \(\mathcal{F}\) Int = Bool
  type instance \(\mathcal{F}\) Bool = Bool
\end{verbatim}

A consequence of this is that a function such as:

\begin{verbatim}
  f :: \(\mathcal{F}\) a \rightarrow \(\mathcal{F}\) a
  f x
  = x
\end{verbatim}

is unusable, since the type \(\mathcal{F} a\) conveys no information about \(a\). This is not the case when using data families:

\begin{verbatim}
  data family \(\mathcal{G}\) a
  data instance \(\mathcal{G}\) Int = IntG Bool
  data instance \(\mathcal{G}\) Bool = BoolG Bool
  g :: \(\mathcal{G}\) a \rightarrow \(\mathcal{G}\) a
  g x
  = x
\end{verbatim}

for calling the function \(g\) will require using either the \texttt{IntG} or \texttt{BoolG} constructors which, being unique to a data type instance, will betray the identity of the type \(a\).
2.7 Constraint kinds, rich kinds and kind polymorphism

Kinds are to types as types are to values. In comparison to its type system, however, Haskell 98’s kind system is comparatively sparse:

\[
\kappa ::= \star \\
\quad \mid \kappa \to \kappa
\]

GHC bolsters this set with the constraint kind Constraint, which is the kind of contexts such as \(\text{Eq } a\) or \(n \sim \text{Zero}\). This handling of contexts means that constraint synonyms and families may be defined in the same manner as type synonyms and families, for example:

\[
\text{type EqNum } a \\
\quad = (\text{Eq } a, \text{Num } a)
\]

However, excluding this enhancement, Haskell 98’s type system remains largely untyped. Revisiting the \(\text{Vec}\) type from Section 2.6, for example:

\[
\text{data Vec } a \ n \ \\n\quad \text{NilV :: Vec } a \ \text{Zero} \\
\quad \text{ConsV :: } a \to \text{Vec } a \ n \to \text{Vec } a \ ((\text{Suc } n))
\]

the \(\text{Vec}\) type constructor has kind:

\[
\text{Vec :: } \star \to \star \to \star
\]

Consequently, it is possible to construct (albeit uninhabitable) types in which the type \(n\) is neither \(\text{Zero}\) or \(\text{Suc } m\), for some \(m\), e.g.:

\[
\bot :: \text{Vec } \text{Int } \text{Bool} \\
\bot :: \text{Vec } \text{Char } (\text{Maybe } \text{Int})
\]

Yorgey et al. describe a suite of enhancements implemented in GHC which enhance Haskell’s kind system [66]. Most notable is the promotion of suitable types and constructors. Recall the empty \(\text{Zero}\) and \(\text{Suc}\) types from Section 2.2, for instance:

\[
\text{data Zero} \\
\text{data Suc } n
\]

Promotion permits us to write a more ‘traditional’ value-level definition of the natural numbers:

\[
\text{data Nat} \\
\quad = \text{Zero} \\
\quad | \text{Suc Nat}
\]

which is then automatically promoted to produce:
• A *type* Zero and a *type constructor* Suc, which replace the previous empty types for working with natural numbers at the type level.

• A *kind* Nat of sort □ (pronounced ‘box’). Zero thus has kind Nat and Suc has kind Nat → Nat.

The use of Zero and Suc in Vec’s definition will result in it now being attributed the kind:

Vec :: * → Nat → *

It is also worth noting that this could also be enforced explicitly by declaring Vec with a *kind signature*, supported by another GHC extension:

```data Vec :: * → Nat → * where
  NilV :: Vec a Zero
  ConsV :: a → Vec a n → Vec a (Suc n)`
```

*Kind polymorphism* arises when a type constructor is promoted. Promoting the trusty list type, for example, yields the following:

```
′[[ ]] :: [κ]
(:) :: κ → [κ] → [κ]
```

where the quote (’) in the name of ′[[ ]] serves to distinguish it from the existing list type constructor [ ]. As one might expect, κ may be instantiated to any kind. Perhaps unexpected is the fact that promotion does not introduce a kind constructor [ ] of sort □ → □ but a set of kinds [κ], each of sort □; this permits a simpler implementation. Note also that the constraining of polymorphic kinds (as would be introduced by promoting GADTs) is not supported at the time of writing. Since GADT constructors introduce coercions between types, their promotion would require the notion of coercions between kinds. Yorgey et al. argue that this would complicate the implementation significantly and defer such enhancements to a future implementation. Note that in the absence of kind coercions, kind equality boils down to the rather simpler idea of α-equivalence.
Embedding domain-specific languages

There has been much work on embedding DSLs and their type systems into other languages. In this chapter we survey several of the techniques used for embedding DSLs in Haskell, with a focus on how well the type system of the DSL in question may be integrated with that of Haskell. Later, we examine some of the research into DSL design and implementation using type-level programming, as well as what has been explored in the presence of even more sophisticated type systems. Note that throughout this thesis we are concerned with embedding DSLs which comprise a pure subset of the host language, here Haskell. For the purposes of consistency we shall base this section around one such expression language, $L$, comprising values, addition, equality and conditional evaluation:

$$e ::= v$$
$$| e + e$$
$$| e ≡ e$$
$$| \text{if } e \text{ then } e \text{ else } e$$

Initially we shall permit only integer and boolean values, though we will later remove this restriction and consider a polymorphic embedded language. As an aside, it should be noted that while the methods described herein are presented approximately in order of increasing sophistication, it is not uncommon to see several approaches combined in the design of a DSL.

3.1 Untyped and phantom-typed representations

Haskell’s ADTs provide arguably the simplest mechanism for defining new types. A first translation of an abstract syntax tree (AST) for $L$ might look as follows:

```
data Value
  = BoolV Bool
  | IntV Int

data Exp
  = ValueE Value
  | AddE Exp Exp
  | EqE Exp Exp
  | CondE Exp Exp Exp
```
or, inlining the definition of the Value type into that of Exp:

```haskell
data Exp
  = BoolE Bool
  | IntE Int
  | AddE Exp Exp
  | EqE Exp Exp
  | CondE Exp Exp Exp
```

where, for example, the expression `if 3 ≡ 4 then 5 + 6 else 7` is represented by the value:

```haskell
e_{good} = CondE (EqE (IntE 3) (IntE 4))
             (AddE (IntE 5) (IntE 6)) (IntE 7)
```

As for a potential consumer of expressions, the compileSM function below implements a compiler for a very simple stack machine that only supports integers; the booleans False and True will be encoded as the integers 0 and 1 in the usual way. Ignoring unique label generation, we have:

```haskell
compileSM :: Exp → String
compileSM (BoolE x)   = "PUSH " ++ show (fromEnum x) ++ "\n"
compileSM (IntE x)    = "PUSH " ++ show x ++ "\n"
compileSM (AddE e1 e2) = compileSM e1 ++ compileSM e2 ++ "ADD\n"
compileSM (EqE e1 e2) = compileSM e1 ++ compileSM e2 ++ "EQ\n"
compileSM (CondE p t f) = compileSM p ++ "CMP 0\n"
                         ++ "BEQ L1\n"
                         ++ compileSM t ++ "B L2\n"
                         ++ "L1: " ++ compileSM f ++ "L2: "
```

Here, we are fortunate that the set of types supported by compileSM is precisely that captured by Exp. If we were unable to compile boolean expressions, for example, we would be forced to omit the clause for BoolE, resulting in a partial function and thus the potential for a run-time exception. Alternatively, consider the AST expression:

```
e_{bad} = CondE (EqE (IntE 3) (BoolE False))
             (AddE (BoolE True) (IntE 4)) (IntE 5)
```

We should expect $e_{bad}$ to be ill-typed: equality is not defined between integers and booleans and addition should only operate over integers. As it stands, however, there is no way to prevent such an error statically – the expressions defined by Exp are completely untyped. In an attempt to rectify this, we might introduce a phantom type (Section 2.2) to the Exp type in order to represent the type of value being manipulated, as in Figure 3.1. Alas, the example expression $e_{bad}$ given earlier is in fact still well-typed:

```
BoolE :: Bool → Exp a
IntE   :: Int → Exp a
```
The limited expressivity of ADTs means that only the CondE constructor has the correct type. In order to prevent the misuse of Exp’s constructors we need to define suitable smart constructors:

\[
\begin{align*}
\text{boolE} &:: \text{Bool} \rightarrow \text{Exp} \text{Bool} \\
\text{boolE} &= \text{BoolE} \\
\ldots \\
\text{addE} &:: \text{Exp} \text{Int} \rightarrow \text{Exp} \text{Int} \\
\text{addE} &= \text{AddE}
\end{align*}
\]

Hiding Exp’s actual constructors from users of the DSL then ensures that only legal expressions may be built. Furthermore, consumers of well-typed terms may freely erase types in order to simplify the processing of expressions. As an example, Nikola (Mainland and Morrisett [36]) is a DSL which targets Nvidia’s range of general-purpose graphical processing units (GPGPUs) using the CUDA programming language, a domain-specific variant of C. Figure 3.2 defines the type of Nikola expressions. DExp is the type responsible for representing the AST corresponding to an expression and is completely untyped. Exp wraps DExp with a phantom type \(a\), thus producing a type amenable to the provision of a type-safe interface. While such an interface is valuable for
ensuring that only well-typed expressions are constructed, it is convenient for the compiler not to be hampered by the presence of DSL types at the Haskell type level. For example, we may wish to represent C parameter lists as lists of expressions. However, it is common for a function to accept parameters of different types, e.g.:

\[
\begin{align*}
p_1 & :: \text{Exp Int} \\
p_2 & :: \text{Exp Float} \\
[p_1, p_2] & :: [\_]\end{align*}
\]

The expression \([p_1, p_2]\) cannot be typed, as a list must hold elements of the same type. Since \text{Int} and \text{Float} are phantom types, the compiler can erase them using the \text{unE} function:

\[
\begin{align*}
\text{[unE } p_1, \text{unE } p_2]\ & :: [\text{DExp}]
\end{align*}
\]

The ability to erase (and introduce) phantom types makes for a simpler implementation of the Nikola compiler, while still guaranteeing a type-safe interface through appropriate smart constructors.

### 3.2 GADTs and tagless encodings

If we are to make \text{Exp} itself type safe without the need for a carefully specified interface of smart constructors, phantom types will not suffice. What is needed is the ability to specify the types of \text{Exp}’s constructors precisely. As seen in Section 2.3, GADTs allow us to do just that. In the case of \text{Exp}, we shall augment the types of its constructors as in Figure 3.3. In keeping with the example of Section 2.3, we have enhanced the type of the EqE constructor to provide an Eq \(a\) context; a pattern-match on EqE will hence introduce the knowledge that values of type \(a\) are at least comparable under equality, even if we do not know what the type \(a\) is. With
this representation, the expression \( e_{bad} \) will be rejected as as ill-typed due to it possessing the following unsatisfiable type:

\[
e_{bad} :: (a \sim \text{Bool}, a \sim \text{Int}) \Rightarrow \text{Exp} \ a
\]

GADTs can also encode more complex type-level properties. Accelerate (Chakravarty et al. [13]) is another DSL that targets CUDA-capable GPGPUs which uses GADTs to enforce the type-safety of its expression types. Figure 3.4 shows the \text{Idx} and \text{OpenExp} types, which together form define the type of Accelerate expressions. A value of type \text{Idx} \( e \ t \) encodes the De Bruijn index of a variable of type \( t \) (as presented by Altenkirch and Reus [1]), where the type \( t \) is also at the same index in the environment \( e \). The environment is represented as a snoc list, in which the last element is accessible in constant time:

\[
\text{data} \ \text{xs} \triangleright x
\]

= \text{xs} \triangleright x

The constructor \text{ZeroIdx} thus possesses a type which states that the type \( t \) is at the head of the list, while \text{SuccIdx} covers the case in which \( t \) is somewhere in the list's tail. By parameterising the \text{OpenExp} type by an environment of types \( e \), the \text{Var} constructor may capture the binding site and type of any variable in a given expression by associating it with an appropriate \text{Idx} value.

It is also possible to rewrite a GADT and any associated semantics as a type class and a set of instances. So-called tagless encodings as described by Pašalić et al. [44] and Carette et al. [10]
data Idx e t where
    ZeroIdx :: Idx (e ▷ t) t
    SuccIdx :: Idx e t → Idx (e ▷ s) t

data OpenExp e a t where
    Var :: ArrayElem t
        ⇒ Idx e (ElemRepr t) → OpenExp e a t
    Const :: Elem t
        ⇒ ElemRepr t → OpenExp e a t
    Pair :: (Elem s, Elem t)
        ⇒ OpenExp e a (ElemRepr s)
        → OpenExp e a (ElemRepr t)
        → OpenExp e a (ElemRepr (s, t))

...
intE \(x\)
\[= \text{CompileSM} \left( \text{"PUSH } + + \text{show } x + + \text{"\n"} \right)\]

daddE \(\text{CompileSM} s_1\) \(\text{CompileSM} s_2\)
\[= \text{CompileSM} \left( s_1 + + s_2 + + \text{"ADD\n"} \right)\]

eqE \(\text{CompileSM} s_1\) \(\text{CompileSM} s_2\)
\[= \text{CompileSM} \left( s_1 + + s_2 + + \text{"EQ\n"} \right)\]

condE \(\text{CompileSM} p\) \(\text{CompileSM} t\) \(\text{CompileSM} f\)
\[= \text{CompileSM} \$
\begin{align*}
p & \leftarrow \text{"CMP #0\n" plus BEQ L1\n" plus t plus} \\
& \text{BR L2\n" plus L1: " plus f plus L2: "}
\end{align*}
\]

The compileSM function itself serves only to pick the correct instance of TaglessExp:

\[
\text{compileSM} :: \text{CompileSM} a \rightarrow \text{String} \\
\text{compileSM} \left( \text{CompileSM} s \right) \\
\quad = s
\]

An unfortunate side-effect of using tagless representations as we have presented is that they restrict the sharing of expressions between multiple semantics. The following excerpt declares a tagless counterpart to the evaluate function given in Section 2.3 and a function which both evaluates and compiles its argument:

\[
\textbf{newtype} \quad \text{Evaluate} \ a \\
\quad = \text{Evaluate} \ a \\
\textbf{instance} \quad \text{TaglessExp} \ \text{Evaluate} \ 	extbf{where} \\
\quad \text{boolE} \ x \quad = \text{Evaluate} \ x \\
\quad \text{intE} \ x \quad = \text{Evaluate} \ x \\
\quad \ldots \\
\text{evaluate} :: \text{Evaluate} \ a \rightarrow a \\
\text{evaluate} \ (\text{Evaluate} \ x) \\
\quad = x \\
\quad f \ e \quad = \ldots (\text{evaluate} \ e) \ldots (\text{compileSM} \ e)
\]

\(f\) will not type check without an explicitly annotated rank-2 type, due to its attempt to instantiate both the CompileSM and Evaluate instances of TaglessExp. Of course, we can provide such a type straightforwardly:

\[
f :: (\forall e. \ \text{TaglessExp} \ e \Rightarrow e \ a) \rightarrow \ldots
\]

Alternatively, we might anticipate the need to provide rank-2 types to all functions which wish to use multiple instances of the TaglessExp class and close our expressions once and for all:
newtype AnyTaglessExp a
    = AnyTaglessExp (\e. TaglessExp e ⇒ e a)

The type AnyTaglessExp boxes a value that may be instantiated to any instance of the TaglessExp class, and now mirrors the Exp GADT defined in Section 2.3. The compileSM and evaluate functions once more pick the appropriate TaglessExp instances, though they must now remove the AnyTaglessExp box also:

\[
\begin{align*}
\text{compileSM} & \colon \text{AnyTaglessExp } a \rightarrow \text{String} \\
\text{compileSM} (\text{AnyTaglessExp} (\text{CompileSM } s)) &= s \\
\text{evaluate} & \colon \text{AnyTaglessExp } a \rightarrow a \\
\text{evaluate} (\text{AnyTaglessExp} (\text{Evaluate } x)) &= x
\end{align*}
\]

### 3.3 Polymorphic languages and parameterised type systems

We will now discuss a polymorphic version of $\mathcal{L}$, in which values may be of any type. Taking the Exp GADT as a starting point, we replace the BoolE and IntE constructors with a single ValueE constructor as in Figure 3.5. ValueE is truly polymorphic, admitting even function values. In generalising AddE, however, we have restricted the set of acceptable types to those that are members of the Haskell Prelude’s Num type class, and are thus in some sense numeric. In doing so we guarantee that the (+) operator is defined over such values, and hence retain the ability to write a type-safe evaluator.

We do not retain the ability to write the compileSM function, however. The problem is that we must now handle a single ValueE constructor instead of the separate BoolE and IntE constructors:

\[
\begin{align*}
\text{compileSM} & \colon \text{Exp } a \rightarrow \text{String} \\
\text{compileSM} (\text{ValueE } x) &= \ldots \\
\ldots
\end{align*}
\]

Previously, it was the case that pattern-matching the BoolE or IntE constructors introduced the knowledge that $x$ was a boolean or integer. Pattern-matching the ValueE constructor only tells us that $x$ has type $a$. Since, there is no way for the compiler to know that values of type $a$ are compilable, we cannot complete the definition of compileSM. One solution is to constrain the ValueE constructor so that it can only be used to lift values which are known to be compilable:

\[
\begin{align*}
\text{class } \text{IntBool } a \text{ where} \\
\text{toInt} & \colon a \rightarrow \text{Int}
\end{align*}
\]
data Exp a where
  BoolE :: Bool → Exp Bool
  IntE :: Int → Exp Int
  AddE :: Exp Int → Exp Int → Exp Int
  EqE :: Eq a ⇒ Exp a → Exp a → Exp Bool
  CondE :: Exp Bool → Exp a → Exp a → Exp a

Figure 3.5: Unifying the BoolE and IntE constructors.

instance IntBool Int where toInt = id
instance IntBool Bool where toInt = fromEnum

data Exp a where
  ValueE :: a → Exp a
  AddE :: Num a ⇒ Exp a → Exp a → Exp a
  EqE :: Eq a ⇒ Exp a → Exp a → Exp Bool
  CondE :: Exp Bool → Exp a → Exp a → Exp a

This is a common approach (used, for example, by Axelsson et al. [6] and Pike et al. [49]): the OpenExp type of Accelerate (Chakravarty et al. [13]) shown earlier supports only those types which are members of the ArrayElem or Elem type classes, both of which are designed to support only types which are usable on a GPU.

However, constraining the ValueE constructor in this manner means that many previously acceptable terms such as ValueE const, for example, are now ill-typed – there is no instance IntBool (a → b → a). During the course of ‘fixing’ the compileSM function the Exp type has been restricted too much. Even a function such as evaluate, which can handle a superset of the terms compilable by compileSM can no longer operate over such values as they are no longer constructible. One way to avoid this problem is to use a technique proposed by Hughes [25], in which types are also parameterised by the constraints which are to be applied to them. Hughes realises constraints as dictionaries, but in the presence of GHC’s constraint kinds (Section 2.7) the following example makes use of constraints as first-class values directly:
With \( \text{Exp} \) (now of kind \( \text{Constraint} \rightarrow \star \rightarrow \star \)) parameterised over the constraints placed upon its values, \( c \), \( \text{compileSM} \) need only pick the desired context:

\[
\text{compileSM} :: \text{Exp } \text{IntBool } a \rightarrow \text{String} \\
\text{compileSM } (\text{ValueE } x) = \text{"PUSH " + show (toInt x) + "\n"}
\]

Under this scheme, the expression \( \text{ValueE } \text{const} \) is still well-typed, although not acceptable as an argument to \( \text{compileSM} \). Such flexibility must be traded for ease of expression and type inference, however. Furthermore, this parameterisation breaks down in the presence of other features of the type system such as higher-ranking types, which we will discuss in detail in Chapter 4.

### 3.4 Name binding and higher-order abstract syntax

The introduction of the polymorphic \( \text{ValueE} \) brought with it the ability to embed arbitrary functions in \( \mathcal{L} \)-expressions, for example:

\[
\text{ValueE } \text{const} :: \text{Exp } (a \rightarrow b \rightarrow a)
\]

or, under Hughes’ scheme:

\[
\text{ValueE } \text{const} :: c (a \rightarrow b \rightarrow a) \Rightarrow \text{Exp } c (a \rightarrow b \rightarrow a)
\]

Observing the essence of what a function provides—namely \textit{binding}—though, is impossible, as in Haskell functions are just values like any other. Therefore, if we are to capture binding in the terms of our language, we must explicitly add constructors to that effect,\(^1\) presumably in a manner that preserves the type safety of our DSL. One approach is to manually handle names and their binding ourselves. Figure 3.6 shows how the \( \text{Exp} \) type might be extended with the constructors \( \text{VarE} \), \( \text{LamE} \) and \( \text{AppE} \) using the types of the \text{Unbound} library presented by Weirich et al. [65], which provides a DSL for expressing the binding semantics of a language. The abstract type \( \text{Name } a \) represents a bound name which may be substituted with a value of type \( a \). The type \( \text{Bind } (\text{Name } a) \ b \) represents a value of type \( b \) in which a name of type \( \text{Name } a \) is bound. Given such a type definition, the library provides a series of combinators for performing substitution, calculating the free variables of a term and so on.

\(^1\)Or class methods in the case of a tagless encoding.
An alternative approach is to use higher-order abstract syntax (HOAS) in which the binding constructs of the host language, here Haskell, are used instead of an explicit reimplementation tailored to the DSL. The Exp type becomes:

```haskell
data Exp a where

  VarE :: Name (Exp a) → Exp a
  LamE :: Bind (Name (Exp a)) (Exp b) → Exp (a → b)
  AppE :: Exp (a → b) → Exp a → Exp b
```

The LamE constructor now lifts a Haskell function. The VarE constructor is no longer needed; terms must make use of Haskell's variables directly, e.g.:

\[
e_{\text{closed}} = \text{LamE} (\lambda x \rightarrow x)
\]
\[
e_{\text{open}} = \text{LamE} (\lambda x \rightarrow y)
\]
\[
e_{\text{exotic}} = \text{LamE} (\lambda x \rightarrow \text{case } x \text{ of } \{ \text{AppE} (\text{LamE } f) y \rightarrow f \ y \}; \_ \rightarrow x})
\]

Here, \(e_{\text{closed}}\) embeds the identity function, while the expression \(e_{\text{open}}\) will be statically rejected by the Haskell compiler, assuming that there is no variable \(y\) in scope. The expression \(e_{\text{exotic}}\) is an exotic term and reflects the fact that the LamE and AppE constructors are not a true realisation of HOAS: since Haskell admits pattern-matching on lambda-bound variables, it is possible to write expressions which do not correspond to any lambda term. One method of preventing the construction of exotic terms is to use parametricity (so-called parametric HOAS or PHOAS), as presented in various forms by Coquand and Huet [17], Washburn and Weirich [64], Atkey [3] and Chlipala [15]:

```haskell
data ExpV v a where

  VarE :: v a → ExpV v a
  LamE :: (v a → ExpV v b) → ExpV v (a → b)
  AppE :: ExpV v (a → b) → ExpV a → ExpV b

newtype Exp a

  = Exp (\forall v. \text{ExpV } v a)
```

Figure 3.6: Using the types of the Unbound library [65] to describe the binding characteristics of the Exp language.
Here, the ExpV type is parameterised by the type of variables which may be bound by lambda expressions, \( \nu \). By using rank-2 polymorphism to abstract this type, the Exp type statically rejects LamE expressions whose functions perform case analysis on their arguments (for such analysis would require knowledge of the argument type). Indeed, the only way in which such arguments may be used is through the VarE constructor.

As the prevention of exotic terms is not our primary concern, we shall once more adopt the simpler ‘pseudo-HOAS’ Exp type presented earlier. Just as terms must use Haskell variables, any function wrapped by LamE may be applied as any other first-class Haskell function; extending the evaluate function to handle Exp’s new constructors is thus relatively straightforward:

\[
\text{evaluate} :: \text{Exp} \ a \to a
\]

\[
\ldots
\]

\[
\text{evaluate} (\text{LamE} \ f) = \text{evaluate} \cdot f \cdot \text{ValueE}
\]

\[
\text{evaluate} (\text{AppE} \ e_1 e_2) = (\text{evaluate} \ e_1) (\text{evaluate} \ e_2)
\]

HOAS is not without its disadvantages, however. Most notable is the fact that, since the equality of two functions is in general undecidable, it is impossible to test whether or not two HOAS expressions represent the same term. To this end Atkey et al. give a type-preserving translation from a HOAS term to a nameless term which uses De Bruijn indices [4]. This affords a language designer the ability to present a HOAS-based interface to the user while allowing the underlying implementation to deal with a simpler representation. Indeed, this is precisely the approach taken by Accelerate (Chakravarty et al. [13]), in which HOAS terms are converted to locally-nameless De Bruijn indexed terms before code generation takes place.

### 3.5 Polyvariadic functions

Polyvariadic functions are those which may be invoked with differing numbers and types of arguments. One example is C’s printf function, which accepts a formatting string and a (possibly empty) set of arguments to be printed according to the given string:

\[
\text{printf}("%c %d %f", \ 'a', 42, 3.14159);
\]

Such functions have shown to be incredibly useful in embedding DSLs, allowing language designers to provide flexible interfaces capable of, for example, both compiling (Mainland and Morrisett [36] and property-testing (Claessen and Hughes [16]) functions of arbitrary arity. It might appear that such functions cannot be defined in the context of Haskell’s type system. Thanks to polymorphism, however, this is not the case. The idea is to construct a type which is polymorphic in its return value, e.g.:

\[
\text{printf} :: \text{String} \to a
\]

If we instantiate \( a \) to a function type \( b \to c \), the printf function may accept an additional argument of type \( b \). If we subsequently instantiate \( c \) to another function type \( d \to e \), printf can
class Printable a where
  printf :: String → a

instance Printable (IO ()) where
  printf s
    = putStrLn s

instance Printable r ⇒ Printable (Char → r) where
  printf s c
    = printf (format s [c])

instance Printable r ⇒ Printable (Int → r) where
  printf s x
    = printf (format s (show x))

format :: String → String → String
format fs v
  = ...

Figure 3.7: Defining C’s printf function in Haskell using type class overloading.

receive yet another argument. When the desired number of arguments has been supplied, the
process is terminated by instantiating a non-functional result type.

3.5.1 Context sensitivity through type class overloading

Deciding the context in which a function is being invoked (i.e. whether or not it must accept an
extra argument or return a result) is achieved using type class overloading. Figure 3.7 defines
printf as a member of the Printable class, along with some example instances. The instance for
IO () is the base case, in which printf has the type:

  printf :: String → IO ()

In this scenario, there are no arguments to be formatted and so printf need only print the given
string to the screen using the Haskell Prelude’s putStrLn function.

The latter two instances are the recursive cases and extend printf with an additional argument
of type Char or Int respectively. In the instance Printable (Int → r), for example (the Char
case follows similarly), printf has the type:

  printf :: Printable r ⇒ String → Int → r

where the type (Int → r) has been unfolded in accordance with the right-associativity of the
type constructor (→). The Int argument (named x in Figure 3.7) is transformed into a String
using the show function before being passed as an argument to the format function. Intuitively,
format is responsible for placing a given string value $v$ into the next position specified in the format string $fs$, i.e.:

$$\text{format } \text{"%c %d %f" } \equiv \text{"a %d %f"}$$

We omit format’s implementation for brevity, but the idea is that, once the argument $x$ has been consumed, the partially completed string is recursively processed by the printf function given in the superclass context, which has type:

$$\text{printf : } \text{Printable } r \Rightarrow \text{String } \rightarrow \text{r}$$

This process continues until the base case’s instance is reached. A complete application might look as follows:

$$\text{main : } \text{IO } () \Rightarrow \text{main}
\text{main}
= \text{printf } \text{"%c %d %f" } \text{'a' } (42 :: \text{Int}) (3.14159 :: \text{Float})$$

Note that the types annotating the values 42 and 3.14159 are required. The unannotated literal 42, for instance, is overloaded and has type $\text{Num } a \Rightarrow a$. While we could write an instance of the form:

$$\text{instance } (\text{Num } a, \text{Printable } r) \Rightarrow \text{Printable } (a \rightarrow r) \text{ where}$$

this will overlap with the existing instances ($\text{Char} \rightarrow r$) and ($\text{Int} \rightarrow r$); recall that the context plays no role in instance selection (Section 2.1).

In the context of our DSL, we can use a polyvariadic function to make the HOAS embedding of functions easier to use. Alone, LamE can only embed an unary function:

$$\text{LamE : } (\text{Exp } a \rightarrow \text{Exp } b) \Rightarrow \text{Exp } (a \rightarrow b)$$

Embedding a function which accepts more than one argument thus requires nested applications of the constructor, as in:

$$\text{LamE } (\lambda x \rightarrow \text{LamE } (\lambda y \rightarrow x)) :: \text{Exp } (a \rightarrow b \rightarrow a)$$

This is both verbose and syntactically noisy. Ideally, we would like a variant of LamE, polyLamE say, which embeds functions of any arity:

$$\text{polyLamE } (\lambda x \rightarrow x) :: \text{Exp } (a \rightarrow a)$$
$$\text{polyLamE } (\lambda x y \rightarrow x) :: \text{Exp } (a \rightarrow b \rightarrow a)$$
$$\text{polyLamE } (\lambda x y z \rightarrow x) :: \text{Exp } (a \rightarrow b \rightarrow c \rightarrow a)$$

We shall define the Variadic class to this end:
class Variadic a r s where
    polyLamE :: (a \rightarrow r) \rightarrow s

The base case instance is as we should expect:

instance Variadic (Exp a) (Exp b) (Exp (a \rightarrow b)) where
    polyLamE = LamE

Here polyLamE has the type (Exp a \rightarrow Exp b) \rightarrow Exp (a \rightarrow b) and must embed an unary function. As mentioned above, this is what the LamE constructor does; we are therefore done. In the recursive case, we shall assume an induction hypothesis that states that there exists an instance Variadic b r (Exp s), for some b, r and s and thus a function:

    polyLamE :: (b \rightarrow r) \rightarrow Exp s

From this, it is required to extend polyLamE such that it embeds functions accepting one extra expression:

    polyLamE :: (Exp a \rightarrow b \rightarrow r) \rightarrow Exp (a \rightarrow s)

Assuming that the function to be embedded is \( f \), of type Exp a \rightarrow b \rightarrow r, providing \( f \) with a value of type Exp a, \( x \) say, will give us a function of type b \rightarrow r which we may pass to the polyLamE function made available through the induction hypothesis:

    polyLamE (f x) :: Exp s

Where does \( x \) come from? We introduce another LamE constructor, which accepts a function of type Exp a \rightarrow Exp b, and abstract \( x \) as the argument:

instance Variadic b r (Exp s) \Rightarrow Variadic (Exp a) (b \rightarrow r) (Exp (a \rightarrow s)) where
    polyLamE f = LamE (\lambda x \rightarrow polyLamE (f x))

Now, LamE’s argument has type Exp a \rightarrow Exp s. The result of applying LamE is thus a value of the desired type Exp (a \rightarrow s).

3.5.2 Inferring types with global and local functional dependencies

As we saw when demonstrating the use of the printf function, polyvariadic functions are susceptible to the need for type annotations due to the need for the compiler to select instances to resolve their types. As it stands, the Variadic class is also a victim in this regard. Consider the application:

AppE (polyLamE const)
Knowing that \( \text{const} \) has the type \( \sigma \to \tau \to \sigma \), we’d like the compiler to infer the type:

\[
\text{AppE} \ (\text{polyLamE} \ \text{const}) :: \text{Exp} \ \sigma \to \text{Exp} \ (\tau \to \sigma)
\]

However, the type is actually inferred as:

\[
\text{AppE} \ (\text{polyLamE} \ \text{const}) :: \text{Variadic} \ \sigma \ (\tau \to \sigma) (\text{Exp} \ (\varphi \to \psi)) \Rightarrow \text{Exp} \ \varphi \to \text{Exp} \ \psi
\]

While it may be clear to us that the constraint \( \text{Variadic} \ \sigma \ (\tau \to \sigma) (\text{Exp} \ (\varphi \to \psi)) \) can be resolved by picking the recursive instance \( \text{Variadic} \ (\text{Exp} \ a) (b \to r) (\text{Exp} \ (a \to s)) \), this is not how constraint resolution proceeds. The first problem is that discussed in Section 2.1, and concerns the ambiguity of the types \( \sigma \) and \( \tau \). Assume that we annotate the expression \( \text{AppE} \ (\text{polyLamE} \ \text{const}) \) with the type we would like inferred:

\[
\text{AppE} \ (\text{polyLamE} \ \text{const}) :: \text{Exp} \ \sigma \to \text{Exp} \ (\tau \to \sigma)
\]

For this to type check, the compiler has to resolve the constraint:

\[
\text{Variadic} \ \alpha \ (\beta \to \alpha) (\text{Exp} \ (\sigma \to \tau \to \sigma))
\]

Note the introduction of the fresh type variables \( \alpha \) and \( \beta \). \( \text{Variadic} \) defines a relation over triples of types: knowing the types \( \sigma \) and \( \tau \) does nothing to help instantiate the types \( \alpha \) and \( \beta \). Moreover, since \( \alpha \) and \( \beta \) do not appear in the type \( \text{Exp} \ \sigma \to \text{Exp} \ (\tau \to \sigma) \), even an explicit type annotation cannot be used to offer such information.

We can fix this problem by introducing a functional dependency to the definition of the \( \text{Variadic} \) class:

\[
\text{class} \ \text{Variadic} \ a \ r \ s \ | \ s \to a \ \text{where}
\quad \text{polyLamE} :: (a \to r) \to s
\]

Now, given an instantiation of the result type \( s \) (here \( \text{Exp} \ (\sigma \to \tau \to \sigma) \)), the type checker may infer the argument type \( a \) for it is uniquely determined by \( s \). The type-annotated application is now well-typed:

\[
\text{AppE} \ (\text{polyLamE} \ \text{const}) :: \text{Exp} \ \sigma \to \text{Exp} \ (\tau \to \sigma)
\]

However, the inferred type remains far from ideal:

\[
\text{const} :: \sigma \to \tau \to \sigma
\]
\[
\text{AppE} \ (\text{polyLamE} \ \text{const})
\quad :: \text{Variadic} \ \tau \ (\text{Exp} \ \varphi) (\text{Exp} \ \psi) \Rightarrow \text{Exp} \ \varphi \to \text{Exp} \ \psi
\]
\textbf{instance} Variadic \((\text{Exp } a) (\text{Exp } b) (\text{Exp } (a \rightarrow b))\) \textbf{where}
\begin{align*}
polyLamE &= \text{LamE} \\
\text{polyLamE } f &= \text{LamE } (\lambda x \rightarrow \text{polyLamE } (f x))
\end{align*}

\begin{array}{ll}
\text{AppE } (\text{polyLamE const}) & \quad \langle \text{Initially} \rangle \\
:: \quad \text{Variadic } \sigma (\tau \rightarrow \sigma ) (\text{Exp } (\varphi \rightarrow \psi )) & \Rightarrow \text{Exp } \varphi \rightarrow \text{Exp } \psi \\
\text{AppE } (\text{polyLamE const}) & \quad \text{FD} (s \rightarrow a) \\
:: \quad \text{Variadic } (\text{Exp } \varphi ) (\tau \rightarrow \text{Exp } \varphi ) (\text{Exp } (\varphi \rightarrow \psi )) & \Rightarrow \text{Exp } \varphi \rightarrow \text{Exp } \psi \\
\text{AppE } (\text{polyLamE const}) & \quad \langle \text{Initially} \rangle \\
:: \quad \text{Variadic } \tau (\text{Exp } \varphi ) (\text{Exp } \psi ) & \Rightarrow \text{Exp } \varphi \rightarrow \text{Exp } \psi \\
\end{array}

Figure 3.8: Type inference in the presence of specialised class instances.
Figure 3.8 shows the steps taken in inferring this type, where the rule \( \text{fD}(f) \) indicates application of the functional dependency \( f \) and a circled number \( n \) denotes the selection of instance \( n \). The problem is that upon reaching the last step, the compiler seeks an instance \( \text{Variadic} \, \tau \, (\text{Exp} \, \varphi) \, (\text{Exp} \, \psi) \). However, both instances require the compiler to unify \( \psi \) with a function type, something it is not prepared to do in order to select an instance.

The trick is to move the constraint that \( \psi \) be a function type from the instance head to the instance context using an equality constraint, as in:

\[
\text{instance} \, (\text{Variadic} \, b \, r \, (\text{Exp} \, s), f \sim (a \rightarrow s))
\Rightarrow \text{Variadic} \, (\text{Exp} \, a) \, (b \rightarrow r) \, (\text{Exp} \, f) \quad \text{where}
\]

The equality constraint \( f \sim (a \rightarrow s) \) is sometimes referred to as a local functional dependency due to the fact that it scopes only over an instance of Variadic rather than the class as a whole. Rewriting the pair of Variadic instances to use local functional dependencies yields the inference chain given in Figure 3.9, where \( \text{eq} \) denotes the resolution of equality constraints. We see that, modulo \( \alpha \)-renaming, we now infer the correct type. In essence, deferring type instantiation to superclass equality constraints tricks the compiler into greedily selecting instances which it would otherwise not consider, facilitating the pervasive type inference we desire.
\begin{align*}
\text{instance } & (f \sim (a \to b)) \\
& \Rightarrow \text{Variadic } (\text{Exp } a) (\text{Exp } b) (\text{Exp } f) \quad \text{where} \\
\text{polyLamE} & = \text{LamE} \\
\text{instance } & (\text{Variadic } b \, r \, t, t \sim \text{Exp } s, f \sim (a \to s)) \\
& \Rightarrow \text{Variadic } (\text{Exp } a) (b \to r) (\text{Exp } f) \quad \text{where} \\
\text{polyLamE } f & = \text{LamE } (\lambda x \to \text{polyLamE } (f \, x))
\end{align*}

\begin{align*}
\text{AppE } (\text{polyLamE const}) & \quad \text{(Initially)} \\
& \quad \, :: \quad \text{Variadic } \sigma (\tau \to \sigma) (\text{Exp } (\varphi \to \psi)) \\
& \quad \Rightarrow \text{Exp } \varphi \to \text{Exp } \psi \\
\text{AppE } (\text{polyLamE const}) & \quad \text{FD}(s \to a) \\
& \quad \, :: \quad \text{Variadic } (\text{Exp } \varphi) (\tau \to \text{Exp } \varphi) (\text{Exp } (\varphi \to \psi)) \\
& \quad \Rightarrow \text{Exp } \varphi \to \text{Exp } \psi \\
\text{AppE } (\text{polyLamE const}) & \quad \text{EQ} \\
& \quad \, :: \quad (\text{Variadic } \tau (\text{Exp } \varphi) t, t \sim \text{Exp } \psi, f \sim (\varphi \to \psi)) \\
& \quad \Rightarrow \text{Exp } \varphi \to \text{Exp } \psi \\
\text{AppE } (\text{polyLamE const}) & \quad \text{FD}(s \to a) \\
& \quad \, :: \quad \text{Variadic } (\text{Exp } \alpha) (\text{Exp } \varphi) (\text{Exp } \psi) \\
& \quad \Rightarrow \text{Exp } \varphi \to \text{Exp } \psi \\
\text{AppE } (\text{polyLamE const}) & \quad \text{EQ} \\
& \quad \, :: \quad (\psi \sim (\alpha \to \varphi)) \\
& \quad \Rightarrow \text{Exp } \varphi \to \text{Exp } \psi \\
\text{AppE } (\text{polyLamE const}) & \quad \text{EQ} \\
& \quad \, :: \quad \text{Exp } \varphi \to \text{Exp } (\alpha \to \varphi)
\end{align*}

Figure 3.9: Type inference in the presence of local functional dependencies.
3.6 Type-level programming

The Variadic class of the previous section demonstrated some of the power offered by multi-parameter type classes and functional dependencies. In this section we examine some of the other ways in which these and more recent features of GHC’s type system (such as type families) may be used to encode a wealth of domain-specific information at the type level.

3.6.1 Type families, type classes and functional dependencies

The technique of using phantom types and smart constructors as discussed in Section 2.2 facilitates a separation of interface and implementation: one can expose a type-safe API to a consumer of the DSL whilst exploiting the benefits of a dynamically typed representation internally. Crossing the boundary between these two components necessitates type erasure, in which any accrued phantom types are discarded.

Consider now a scenario in which both sides of the boundary use typed representations, but where there is a mismatch between the two type systems. Kansas Lava (Gill et al. [23, 24]) is an extension of the hardware design language Lava (Bjesse et al. [7]). Kansas Lava supports the definition of both combinatorial and sequential circuits, though the more general abstraction of a Signal forms the basis for the language’s interface:

```haskell
class Signal s where
    liftSig_0 :: Comb a → s a
    liftSig_1 :: (Comb a → Comb b) → s a → s b
    ...
```

Comb is the type of combinatorial circuits. The $\text{liftSig}_n$ family of functions embodies the fact that any combinatorial circuit (or function thereof) can be lifted into the combinatorial fragment of the domain of sequential circuits.

Now suppose we wish to write a circuit which encodes a half adder (where the booleans $\text{True}$ and $\text{False}$ will be used to encode the bits $1$ and $0$). Which of these types should we pick?

```haskell
halfAdder :: (Comb Bool, Comb Bool) → (Comb Bool, Comb Bool)
halfAdder :: Comb (Bool, Bool) → Comb (Bool, Bool)
```

The first preserves pattern-matching:

```haskell
halfAdder (b₁, b₂) = ...
```

while the second may be used with the members of the Signal class ($\text{liftSig}_1$, for example). The solution adopted by Kansas Lava is to use a type family to move between the two representations. Figure 3.10 shows both the Pack class with the associated family Unpacked and an
class Signal s ⇒ Pack s a where
  type Unpacked s a
  pack :: Unpacked s a → s a
  unpack :: s a → Unpacked s a

instance (Wire a, Wire b, Signal s) ⇒ Pack s (a, b) where
  type Unpacked s (a, b) = (s a, s b)

Figure 3.10: The Pack class of Kansas Lava, along with an instance for packing and unpacking pairs.

data HNil = HNil
data HCons a as = HCons a as
type a ::*: as = HCons a as
  x .*: xs = HCons x xs
l :: Char ::*: Bool ::*: Int ::*: HNil
l = ‘a’ .*: False .*: 42 .*: HNil

Figure 3.11: The types of the HL1st library [33], along with an example heterogeneous list.

instance for the pair type constructor (.,).² Such an instance allows us to write a halfAdder function which can be lifted by liftSig₁ without sacrificing pattern-matching:

halfAdder :: Comb (Bool, Bool) → Comb (Bool, Bool)
halfAdder bs
  = case unpack bs of
     (b₁, b₂) → pack (xor₂ b₁ b₂, and₂ b₁ b₂)

As mentioned in Section 2.6, there are clear similarities between type synonym families and multi-parameter type classes annotated with functional dependencies. Indeed, many works have solved similar problems to that encountered by Gill et al. [24] using type classes. HL1st (Kiselyov et al. [33]) is a library for manipulating heterogeneous lists in Haskell as, for example, in Figure 3.11. Heterogeneous lists (and the types thereof) are constructed with HNil and HCons. The right-associative type-level and value-level operators (::*) and (.*.) alias HCons and make for more concise definitions.

²The superclass constraints Wire a and Wire b are unimportant here and are included only for the sake of correctness.
HLlist defines a great many operations over heterogeneous lists at both the type and value level, including higher-order operations such as mapping and reduction. Each is defined using one or more multi-parameter type classes possessing appropriate sets of functional dependencies. For example, the HAppend type class defines the hAppend function, which is responsible for concatenating two heterogeneous lists:

```haskell
class HAppend as bs cs | as bs → cs where
  hAppend :: as → bs → cs

instance HList bs ⇒ HAppend HNil bs bs where
  hAppend HNil ys = ys

instance (HList as, HAppend as bs cs) ⇒ HAppend (HCons a as) bs (HCons a cs) where
  hAppend (HCons x xs) ys = HCons x (hAppend xs ys)
```

The functional dependency \( as \rightarrow bs \rightarrow cs \) states that, given a type \( as \) and a type \( bs \), the type \( cs \) may be determined uniquely. In essence, \( cs \) is a function of \( as \) and \( bs \). That is to say, the HAppend class could be rewritten as:

```haskell
class HAppend as bs where
  type HAppendResult as bs
  hAppend :: as → bs → HAppendResult as bs
```

where \( cs \) is replaced by the application of the type-level function HAppendResult to \( as \) and \( bs \). It is not always possible to rewrite functional dependencies in this manner, however. Sackman and Eisenbach [50] make heavy use of multi-parameter type classes in their embedding of session types into Haskell’s type system, defining even data structures such as hash tables at the type level. In doing so, they define the class of pairs of session types which are dual to each other:

```haskell
class Dual a b | a → b, b → a
```

Here, knowing either of the variables \( a \) or \( b \) is enough to deduce the other: Dual embodies an injective type family. As noted in Section 2.6, it is not currently possible to specify whether or not a type family is injective. For this reason multi-parameter type classes and functional dependencies are still common in type-level programming.

### 3.6.2 Encoding properties with empty phantom types

Section 3.1 presented phantom types as a lightweight means of typing expressions. Their applications are far more widespread however. Potential (Carstens [11]) is a DSL for writing 64-bit
x86 assembly code which makes heavy use of phantom types to provide strong static guarantees about embedded programs. For example, a function for incrementing a number might be written in Potential as follows:

```haskell
increment
  = defun "increment" $ do
    isCode
    mov rdi rax
    loadInt 1 rbx
    add rbx rax
    ret
```

Potential programs follow the System V AMD64 application binary interface [39] (ABI) in which a function’s first argument is passed in the 64-bit register rdi and a result must similarly be placed in 64-bit register rax. The program thus copies the argument to rax. Next, the constant 1 is loaded into the 64-bit register rbx, before a destructive addition overwrites the value in rax with its successor.

As the `do`-notation suggests, Potential programs are monadic computations. Specifically, the functions `isCode`, `loadInt`, `add` and `ret` produce results whose types have the form `PState f p q a`, where `PState` is an indexed monad (as considered by Atkey [2], Uustalu [58], Wadler and Thieman [63] and Filliâtre [20], among others):

```haskell
class IxMonad m where
  return :: a -> m f p p q
  (>>=) :: m f p q a -> (a -> m f q r b) -> m f p r b
```

Note that the type `f` does not strictly form part of the definition of an indexed monad; we shall discuss its role shortly. The indices `p` and `q` in the type `m f p q a` of a monadic computation can be thought of as the action’s pre- and post-conditions, not dissimilar to those of a Hoare triple. The `return` function lifts a pure value into a monadic context whose pre- and post-conditions are one and the same, while `(>>=)` (pronounced ‘bind’) connects the post-condition of an initial monadic action to the pre-condition of a second, creating an action possessing the initial and latter pre- and post-conditions. Of course, the notion of pre- and post-conditions is purely intuitive; each is just a type:

```haskell
data Rax
  rax :: PState f rs (Rax :*: rs) Word64
```

The intention here is that the `rax` action’s type encodes the fact that the register of the same name has been used. This is done by prepending the empty type `Rax` to the ‘pre-condition’ type-level list `rs`. Here we have used the `(:*:)` constructor of the HLisr library, but we could have achieved this equally well using the promoted version of Haskell’s list data type (as we shall demonstrate
in Chapter 4). We note that the rax action is not actually defined as such in Potential; the above type is merely introduced here as an example of the usefulness of indexed monads.

Let us now reveal the mysteries of the type variable \( f \). As mentioned, the rax function above doesn’t appear in Potential, not least because Potential’s type signatures carry more information than simply which registers are used in a computation. Operand size, for example, is also encoded at the type level:

```haskell
class HasSZ d s
```

where an instance `HasSZ d s` states that a value of type \( d \) has size \( s \) (\( s \) being a type-level natural number). As one might expect, this leads to types such as:

```haskell
x :: HasSZ d (Suc (Suc Zero)) ⇒ PState f (... d ...) q ()
```

Unfortunately, the constraints in such types sometimes get in the way. Consider a `render` function, whose job is to generate code for a given program:

```haskell
render :: PState f p q () → String
```

We cannot use `render` to generate code for the expression `x` above. `render`’s type mentions only the unconstrained pre- and post-conditions `p` and `q`, making no reference to the type `d` whose size is to be constrained. The solution is to only enforce the `HasSZ` constraints during expression construction, discarding them during rendering — in effect the provision of optional class constraints. The `HasSZ` class becomes one of a family:

```haskell
data ClassConstraintsOn

data ClassConstraintsOff

class MaybeHasSZ d s f

instance HasSZ d s ⇒ MaybeHasSZ d s ClassConstraintsOn

instance MaybeHasSZ d s ClassConstraintsOff
```

The `MaybeHasSZ` class possesses two instances predicated on the type of the flag `f`. If `f` is instantiated to `ClassConstraintsOn`, the first instance will be selected and the superclass constraint `HasSZ d s` brought into effect. In the even that `f` unifies with `ClassConstraintsOff`, the latter instance will be selected, requiring no such context. The `render` function now has the type:

```haskell
render :: PState ClassConstraintsOff p q () → String
```

Upon passing the value `x` to `render`, the type `f` will become `ClassConstraintsOff` and the hindrance of the `HasSZ` context will be avoided.
Template metaprogramming and quasiquotation

Template Haskell (TH) (Sheard and Peyton Jones [53]) is a template metaprogramming framework for Haskell implemented in GHC. Unlike C++'s templates, for example, template code is written in the same language as the program (i.e. Haskell); in essence, TH makes it possible to call Haskell functions at compile-time. We briefly cover the features of TH that expose this functionality.

Code as data

As with Lisp's S-expressions, Haskell code is represented by a set of ADTs (Figure 3.15). Code generation and manipulation is performed by ordinary Haskell functions of the appropriate type:

\[
tmap :: \text{Int} \rightarrow \text{ExpQ} \\
tmap n = \text{do} \\
\quad f \leftarrow \text{newName} \text{ “f”} \\
\quad xs \leftarrow \text{replicateM} n (\text{newName “x”}) \\
\quad \text{lamE} [\text{varP} f, \text{tupP} (\text{map varP} x)] \\
\quad \quad (\text{tupE} (\text{map} (\text{appE} (\text{varE} f) \cdot \text{varE} x)) \cdot \text{xs})
\]

The application \( tmap \ n \) generates an AST for a function capable of mapping a function over tuples of length \( n \). For example, \( tmap \ 3 \) would generate an AST for the following function:

\[
\lambda f (x_1, x_2, x_3) \rightarrow (f x_1, f x_2, f x_3)
\]

\( tmap \)'s result type \( \text{ExpQ} \) is a synonym for the type \( Q \text{ Exp} \), where \( Q \) is the ‘quotation’ monad, here used for its fresh name generation (the \( \text{newName} \) function). The functions \( \text{lamE}, \text{varP}, \text{tupP} \) and the like are the liftings of their constructor siblings into the \( Q \) monad, e.g.:

\[
\text{LamE} :: [\text{Pat}] \rightarrow \text{Exp} \rightarrow \text{Exp} \\
\text{lamE} :: [\text{PatQ}] \rightarrow \text{ExpQ} \rightarrow \text{ExpQ}
\]

Quotation

The \( Q \) monad is so named due to its providing a means to *quote* Haskell code:

\[
\left\llbracket \lambda x \ y \rightarrow x \right\rrbracket
\]

The above expression is known as a quotation and has type \( \text{ExpQ} \), a monadic computation which when run produces the AST:
By default, Template Haskell supports the quoting of expressions ([...]), types ([t] [...]), and declarations ([d] [...]). **Quasiquoters** as introduced by Mainland [35] are a means of embedding domain-specific syntax into Haskell, whereby a quoted string is translated at compile-time by a function (a quasiquoter) into Template Haskell expressions. Quasiquoters are user-defined and unlike builtin quotations may be used in patterns also (providing a value of type PatQ). Mainland, for example, defines a quoter cfun for embedding C function declarations in Haskell code as-is:

\[
\text{add \, n} = \text{[cfun} | \text{int} \text{add (int \, x)} \{ \text{return \, x +$int\,n;} \} ]
\]

The token $int\,n$ denotes an **anti-quotation** and is a feature of the cfun quasiquoter as opposed to quasiquotation in general. As expected, it signifies that the generated code for the C function add should reference the value of the Haskell value \(n\).

**Splicing**

**Splicing** is the means by which functions may be called at compile-time, and is initiated by the $(...) operator. Splices may be used in various parts of a program, provided the expression being spliced has a type appropriate to the context. For example, in:

\[
\text{ds} = f \, x
\]

\[
\text{ds \, must \, return \, a \, declaration \, whilst \, g \, must \, be \, of \, type \, Int \to ExpQ}.^3
\]

Note that the types

---

3. \(ds\) is actually free to give a list of declarations, having type Q [Dec] (or DecsQ).
of the program fragments represented by $ds$ and $g \, x$ are checked at the point of splicing: while it is possible to construct ASTs representing ill-typed expressions, it is not possible to splice them into Haskell programs.

### 3.8 Dependently-typed languages

In a dependently-typed language, types may depend upon values. Languages falling into this category such as *Epigram* (McBride and McKinna [40]) and *I\(\text{DRIS}*) (Brady [8]) thus possess type systems whose features subsume many of those discussed in Chapter 2, and have consequently found applications in embedding DSLs. *I\(\text{DRIS}*) in particular has been designed with DSL embedding in mind, providing powerful features such as user-defined syntax:

```
syntax if [p] then [t] else [f] = CondE p t f
```

Here, the brackets ([]) denote non-terminals: given an *I\(\text{DRIS}*) equivalent of the Exp type presented throughout this chapter, this statement defines the familiar *if-then-else* statement in terms of the CondE constructor. Moreover, assuming a variant of Exp that embeds functions using De Bruijn indices rather than HOAS, the declaration:

```
(dsl exp
  lambda = LamE
  variable = VarE
  index_first = Zerol
  index_next = SucI
)
```

permits one to write:

```
exp (\lambda x \, y \Rightarrow x + y)
```

and have variable binding and use desugared to applications of the LamE and VarE constructors, with De Bruijn indices generated and consumed through the Zerol and SucI constructors. *I\(\text{DRIS}*) also supports popular Haskell constructs such as *do*-notation and idiom brackets (McBride and Paterson [41]) (as implemented in SHE, discussed shortly), two features which when combined with the syntactical aspects of the language can produce very elegant DSL libraries (Brady and Hammond [9], Brady [8]).

The second incarnation of Epigram is written using the *Strathclyde Haskell Enhancement* (SHE), a Haskell preprocessor that simulates dependently-typed programming in Haskell. More bluntly, SHE implements a Haskell DSL for writing a subset of dependently-typed programs. SHE’s take on the faithful Vec type is as follows:

```
data Vec :: \* \rightarrow \{ \text{Nat} \} \rightarrow \* where
  NilV :: Vec a \{ \text{Zero} \}
  ConsV :: a \rightarrow Vec a \{ n \} \rightarrow Vec a \{ \text{Suc} \, n \}
```

47
Vec exemplifies one of SHE’s primary additions to Haskell: types indexed by data. Analogous to the GHC extensions described in Section 2.7, SHE promotes braced data types and constructors {Nat} and {Zero}, for instance for use at the kind and type levels (strictly speaking, only constructors are promoted; types are simply erased to kind *). Indeed, SHE helped motivate the first-class support for rich kinds and promotion that is now available.

SHE also supports dependent (or Π-) types through singletons. Consider a vector analogue of the Haskell Prelude’s list-constructing replicate function:

\[
\text{replicate} \mathit V :: \forall a. \Pi (n :: \text{Nat}). a \rightarrow \text{Vec} a \{ n \}
\]

\[
\begin{align*}
\text{replicate} \mathit V \{ \text{Zero} \} x &= \text{NilV} \\
\text{replicate} \mathit V \{ \text{Suc} n \} x &= \text{ConsV} x (\text{replicate} \mathit V \{ n \} x)
\end{align*}
\]

In a manner resembling that of the dictionary-passing transformation, the Π-type is desugared to an ordinary function argument:

\[
\text{replicate} \mathit V :: \forall a n. \text{SheSingleton Nat} n \rightarrow a \rightarrow \text{Vec} a n
\]

SheSingleton is a data family which defines a set of runtime witnesses, known as singletons, for each type of a given kind. For Nat SHE defines the instances:

\[
\begin{align*}
\textbf{data family} & \quad \text{SheSingleton} a :: \star \rightarrow \star \\
\textbf{data instance} & \quad \text{SheSingleton Nat} :: \star \rightarrow \star \quad \textbf{where} \\
& \quad \text{ZeroS} :: \text{SheSingleton Nat} \{ \text{Zero} \} \\
& \quad \text{SucS} :: \text{SheSingleton Nat} \{ n \} \rightarrow \text{SheSingleton Nat} \{ \text{Suc} n \}
\end{align*}
\]

The once braced arguments to replicateV now become bona-fide values which fix the type of \( n \) by virtue of the equality constraints they introduce when pattern-matched:

\[
\begin{align*}
\text{replicate} \mathit V \text{ZeroS} x &= \text{NilV} \\
\text{replicate} \ (\text{SucS} n) x &= \text{ConsV} x (\text{replicate} \mathit V n x)
\end{align*}
\]

As with promotion, SHE has inspired subsequent work in this field. Eisenberg and Weirich [19] utilise the support for promotion and rich kinds in building the singletons library, which amongst other features provides Template Haskell functions (Section 3.7) for generating singleton witnesses for types.

Agda (Norell [42]) is another dependently-typed functional language that has shown itself amenable to DSL design thanks to its mixfix syntax. The if-then-else expression bound above, for example, could be written as follows in Agda:
The function `if_then_else_` defines a ternary mixfix operator whose arguments are designated by the underscores (_ _) in its name. While the function in fact accepts four arguments, the first, `A`, is implicit (denoted by the braces in its specification `{ A : Set }`) – the type checker will infer `A` from a given use of the `if_then_else_` operator. For example, the application:

```agda
if True then Zero else Suc Zero
```

results in the type `A` being instantiated to the type `Nat`. In Chapter 4 we shall use Agda in the context of our work; we shall thus give a very brief introduction to the language here. In what follows we present an adaptation of Oury and Swierstra [43]'s embedding of relational algebra (c.f. the Haskell library developed by Leijen and Meijer [34]), which serves particularly well to highlight the properties which may be captured so neatly with full-spectrum dependent types. The definitions:

```agda
data Nat : Set where
  Zero : Nat
  Suc : Nat → Nat

  _+_ : Nat → Nat → Nat
  Zero  + n = n
  Suc m + n = Suc (m + n)
```

do as we should expect, defining the type of natural numbers and an addition function. Unlike Haskell, type signatures in Agda are introduced by a single colon (:). The term `Nat : Set` states that the `Nat` type itself has type `Set`, where `Set` may be likened to Haskell’s kind `⋆` insofar as it being the type of ‘small’ types.

While data types in Agda generally resemble Haskell’s GADTs, Agda makes explicit the distinction between parameterisation and indexing:

```agda
data Vec (A : Set) : Nat → Set where
  [] : Vec A Zero
  _∷_ : { n : Nat } → A → Vec A n → Vec A (Suc n)
```

The signature `T σ₁ σ₂ ... σₘ : τ₁ → τ₂ → ... → τₙ` declares a type `T` which, when parameterised by a set of types `σ₁, σ₂, ..., σₘ` yields a family of types which may be indexed by the types `τ₁, τ₂, ..., τₙ`. The codomains of `T`’s constructors are then permitted to vary only in the indexes `τ₁, τ₂, ..., τₙ`; the parameters `σ₁, σ₂, ..., σₘ` may not influence whether or not the type is habitable, as one might expect.

For example, `Vec` is parameterised by a type `A` of type `Set` and indexed by a natural number. The parameter `A` appears uniformly in the result types of its constructors `[]` and `_∷_`, while the
index of type Nat is instantiated to Zero in the case of [] and the successor of a natural number in the type of _∷_. As seen when illustrating SHE, _∷_ (there ConsV) possesses a dependent (or II-) type: its first argument is a natural number named n which is subsequently used in specifying the remainder of the type.

In a dependently-typed language, the type checker might need to evaluate functions at the type level. For example, given a function for concatenating two vectors:

\[
\begin{align*}
\text{data } \text{Vec} & : \text{Set} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
\text{Vec} & : \text{U} \rightarrow \text{Nat} \rightarrow \text{U}
\end{align*}
\]

the use of the addition function _+_ in the type Vec A (m + n) must result in a type-level computation lest types such as Vec A 3 and Vec A (1 + 2) be thought different. In order to guarantee the type checker’s termination, then, Agda requires that all functions be total. Of course, functions may not only range over values – just as one may use values in types, one can write functions from values to types:

\[
\begin{align*}
\text{data } \text{U} : \text{Set} & \text{ where} \\
\text{Bool} & : \text{U} \\
\text{Char} & : \text{U} \\
\text{Nat} & : \text{U} \\
\text{Vec} & : \text{U} \rightarrow \text{Nat} \rightarrow \text{U}
\end{align*}
\]

The pair (U, decode) is known as a universe. Universes are comparable with Haskell’s type classes in that they specify a set of types. Moreover, they may be used to facilitate overloading:
show \{Nat_\mathcal{U}\} \text{Zero} = \text{“Zero”}
show \{Nat_\mathcal{U}\} (\text{Suc } n) = \text{“Suc ” + parens (show } n) 
show \{\text{Vec}_\mathcal{U} u \text{Zero}\} [] = \text{“[]”}
show \{\text{Vec}_\mathcal{U} u (\text{Suc } n)\} (x :: xs) = parens (show } x) + “ :: ” + parens (show } xs)
parens : \text{String} \rightarrow \text{String} 
parens s = \text{“(” } + s + \text{“)”}

where the pattern \{ p \} matches the pattern p against an implicit argument. Unlike type classes, universes are \textit{closed} after the completion of their definition – new types may not be added to a universe at a later point. In contrast, type classes can be extended freely with instances.

The power of universes (and dependent types as a whole) becomes apparent when one starts to embed a type system. The process is very direct; in the case of the relational algebra described by Oury and Swierstra [43] we begin with:

\begin{align*}
\text{Attribute} & : \text{Set} \\
\text{Attribute} & = \text{String} \times \mathcal{U} \\
\text{Schema} & : \text{Set} \\
\text{Schema} & = \text{List Attribute}
\end{align*}

This states that a schema comprises a list of attributes, each of which is a pair consisting of a name (represented by a String) and a type which is a member of the universe \mathcal{U}. An example schema might be:

\begin{align*}
\text{Users} & : \text{Schema} \\
\text{Users} & = (\text{“Login”, Vec}_{\mathcal{U}} \text{Char}_{\mathcal{U}} 8) :: (\text{“Name”, Vec}_{\mathcal{U}} \text{Char}_{\mathcal{U}} 24) :: \\
& \quad \text{(“Active”, Bool}_{\mathcal{U}}) :: []
\end{align*}

The type of relational algebra expressions, \mathcal{R} is indexed by a schema describing the shape of the data being manipulated:

\begin{align*}
\textbf{data} \quad \mathcal{R} & : \text{Schema} \rightarrow \text{Set} \quad \textit{where} \\
\text{Read} & : \{ s : \text{Set} \} \rightarrow \text{Handle } s \rightarrow \mathcal{R} s \\
\text{Union} & : \{ s : \text{Set} \} \rightarrow \mathcal{R} s \rightarrow \mathcal{R} s \rightarrow \mathcal{R} s \\
\text{Diff} & : \{ s : \text{Set} \} \rightarrow \mathcal{R} s \rightarrow \mathcal{R} s \rightarrow \mathcal{R} s \\
\text{Product} & : \{ s s’ : \text{Set} \} \rightarrow \{ \text{So (disjoint } s s’) \} \rightarrow \mathcal{R} s \rightarrow \mathcal{R} s’ \rightarrow \mathcal{R} (s + s’) \\
\text{Project} & : \{ s : \text{Set} \} \rightarrow (s’ : \text{Schema}) \rightarrow \{ \text{So (s’ } \subseteq s) \} \rightarrow \mathcal{R} s \rightarrow \mathcal{R} s \\
\text{Select} & : \{ s : \text{Set} \} \rightarrow \text{Exp } s \text{ Bool} \rightarrow \mathcal{R} s \rightarrow \mathcal{R} s
\end{align*}
While this definition facilitates type safety through the indexing of both the algebra and the backing database (the abstract Handle type) by the schema of interest, the real magic results from the So type:

```haskell
data So : Bool → Set where
  Oh : So True
```

So’s only constructor is Oh, which is indexed by the boolean value True. There are no values of So False. Consequently, in the event that \( s \) and \( s' \) are not disjoint schema in an application of the Product constructor, say, we will be required to provide a value of type So False – an impossible task! Thus we are statically prevented from trying to create the cartesian product of schema for which it does not make sense to do so.

### 3.9 Discussion

In this chapter we have examined some of the ways in which domain-specific languages can be embedded in a type-safe manner, both in Haskell and full-spectrum dependently-typed languages such as Agda. However, while we have seen that embedding a DSL’s type system into Haskell’s is possible, we encountered difficulties when considering DSLs which target multiple differently-typed implementations. These difficulties are the subject of the next chapter, in which we discuss how multiple type systems can be attributed to a DSL without sacrificing desirable features such as type inference and low syntactic overhead.
Deconstraining data types

The DSLs shown in the previous chapter were all designed with a primary target in mind: Accelerate (Chakravarty et al. [13]) and Nikola (Mainland and Morrisett [36]) both target CUDA, while Potential (Carstens [11]) generates assembly code, for example. In this chapter we will consider DSLs which must target multiple back ends, possibly simultaneously in a single context; we call such DSLs heterogeneous. In doing so we shall see how static type safety may restrict the flexibility of a DSL, and develop the techniques presented in the paper ‘Deconstraining DSLs’ [28] in order to overcome these limitations. Additionally, we extend the work of the paper with a proof that functions constrained using our framework are total.

4.1 The type restriction problem revisited

Recall the polymorphic DSL introduced in Section 3.3:

```
data Exp a where
    ValueE :: a -> Exp a
    AddE :: Num a => Exp a -> Exp a -> Exp a
    EqE :: Eq a => Exp a -> Exp a -> Exp Bool
    CondE :: Exp Bool -> Exp a -> Exp a -> Exp a
```

The task once more is to construct a compiler, compileSM, for a stack machine that supports only integer values; the problem is that Exp’s ValueE constructor lifts values of arbitrary type. An initial solution involved constraining ValueE’s type so as to only admit values of type Int or Bool (an approach taken by Accelerate, for example):

```
class IntBool a where
toInt :: a -> Int

instance IntBool Int where toInt = id
instance IntBool Bool where toInt = fromEnum

data Exp a where
    ValueE :: IntBool a => a -> Exp a

...```

However, this prevents the use of terms such as ValueE const, which makes perfect sense in the context of, say, an evaluator for the DSL.
evaluate :: Exp a → a
evaluate (ValueE x) = x
evaluate (AddE e₁ e₂) = evaluate e₁ + evaluate e₂
evaluate (EqE e₁ e₂) = evaluate e₁ ≡ evaluate e₂
evaluate (CondE p t f) = if evaluate p
then evaluate t
else evaluate f

Hughes [25] suggests that such 'restricted data types' be constrained parametrically by abstracting away the desired context as a parameter of the Exp type:

```haskell
data Exp c a where
  ValueE :: c a ⇒ a → Exp a
  ... 
  compileSM :: Exp IntBool a → String
  compileSM (ValueE x)
  = “PUSH ” ++ show (toInt x) ++ “\n”
```

A value such as ValueE const is now once more well-typed:

```
  ValueE :: c (a → b → a) ⇒ Exp c (a → b → a)
```

but it can not be passed to the compileSM function due to there being no suitable instance IntBool (a → b → a). This appears to solve the problem. We may, for example, make use of both the compileSM and evaluate functions in a single context:

```
evaluate :: Exp c a → a
  ...
f :: Exp IntBool a → ...
f e = ... (compileSM e) ... (evaluate e) ...
```

However, this only works because evaluate's type does not require any particular constraint c; both functions thus agree on e having type Exp IntBool a (indeed, the compiler would have inferred this type had we omitted a signature for f). A case that is unfortunately not so easily solved arises when two different contexts are required. As an example, a pretty-printer might make use of the Haskell Prelude's Show class:

```
pretty :: Exp Show a → String
  ...
f :: Exp c a → ...
f e = ... (compileSM e) ... (pretty e) ...
```

For f to be well-typed, e must possess a type that can be instantiated to both Exp IntBool a and Exp Show a. The usual solution is to introduce a higher-rank type:
While \( f \) will now type check, we cannot apply it. The expression:

\[
 f \ (\text{ValueE False})
\]

is ill-typed. The term \( \text{ValueE False} \) cannot be assigned the type \( \forall c. \ \text{Exp} \ c \ \text{a} \) since without knowing the exact constraint \( c \) \( \text{GHC} \) is unable to satisfy itself that there is an instance \( c \ \text{Bool} \).

We might try:

\[
 f :: (\forall c. \ c \ \text{a} \Rightarrow \text{Exp} \ c \ \text{a}) \rightarrow \ldots
 f \ e = \ldots (\text{compileSM} \ e) \ldots (\text{pretty} \ e) \ldots
\]

But now \( f \) won’t type check unless we explicitly introduce the very constraints we wish to instantiate \( c \) to, namely:

\[
 f :: (\text{IntBool} \ a, \text{Show} \ a) \Rightarrow (\forall c. \ c \ \text{a} \Rightarrow \text{Exp} \ c \ \text{a}) \rightarrow \ldots
 f \ e = \ldots (\text{compileSM} \ e) \ldots (\text{pretty} \ e) \ldots
\]

which defeats the point of abstraction in the first place! What is needed is a method of delaying the need for a platform-specific constraint until the point at which it is picked.

### 4.2 Generic constraints

The techniques presented thus far encode a target platform as a type class constraint which must be satisfied by a given expression type. In doing so, they harness the Haskell type checker’s ability to test the satisfaction of such constraints, but fall foul of its eagerness to do so. As mentioned in this thesis’ introduction, what is actually needed is a way to identify individually the set of features supported by a platform or the set of features present in an expression. This suggests two sets of type class constraints: those which consume feature sets (platform constraints) and those which produce or describe feature sets (expression constraints). Since the Haskell type checker has no means of reasoning about the compatibility of constraints which are defined independently of one another, it will be up to us to check whether or not a set of consumers and a set of producers are compatible. However, this control is precisely what we need: in choosing when to show the type checker how these constraints may be exchanged we may bring about the required ‘delay’ in platform-specific type restriction.

In the case of our running example, the idea is to associate each value of type \( \text{Exp} \) with a list of types \( as \) which represents symbolically the types that appear in the expression. Individual, concrete, constraints (such as \( \text{IntBool} \)) are then replaced with a set of generic constraints of the form \( a \in as \). For the \( \text{ValueE} \) constructor, a single generic constraint \( a \in as \) captures the notion that the type of object wrapped by \( \text{ValueE} \) must be an element of the list of types, \( as \). Similarly for the \( \text{CondE} \) case, which also requires the constraint \( \text{Bool} \in as \) to cater for the type of the
The AddE and EqE cases again follow suit, except that they also retain the original Num a and Eq a constraints. In many, if not all, cases it may suffice to generate these (∈) constraints mechanically, e.g. using Template Haskell, but for the time being we decorate the constructor types explicitly:

**data Exp as a where**

- **ValueE** :: \((a \in as) \Rightarrow a \rightarrow Exp as a\)
- **AddE** :: \((\text{Num } a, a \in as) \Rightarrow Exp as a \rightarrow Exp as a \rightarrow Exp as a\)
- **EqE** :: \((\text{Eq } a, a \in as, \text{Bool } \in as) \Rightarrow Exp as a \rightarrow Exp as a \rightarrow Exp as \text{Bool}\)
- **CondE** :: \((a \in as, \text{Bool } \in as) \Rightarrow Exp as \text{Bool} \rightarrow Exp as a \rightarrow Exp as a \rightarrow Exp as a\)

Importantly, this definition will suffice for all functions defined over the DSL. Note also that Haskell’s type inference engine automatically deletes duplicate constraints, as illustrated by the following expression:

\[ e_{\text{mixed}} = \text{CondE} (\text{EqE} (\text{ValueE } (3 :: \text{Int})) (\text{ValueE } (4 :: \text{Int}))) \]
\[ (\text{ValueE } (0.0 :: \text{Float})) (\text{ValueE } (3.9 :: \text{Float})) \]

The constraint \(\text{Bool } \in as\) is inferred twice: once when typing the subexpression \(\text{EqE} \ldots\) which computes the constraints \((\text{Eq } \text{Int, } \text{Int } \in as, \text{Bool } \in as)\), and once when typing the conditional \(\text{CondE} \ldots\) which independently imposes the constraint \(\text{Bool } \in as\) in addition to \(\text{Float } \in as\) (the top-level result is an \(\text{Exp as Float}\)). The \(\text{Eq } \text{Int}\) constraint is trivially satisfied, and so is removed, and the duplicate \(\text{Bool } \in as\) constraints are collapsed into one. The resulting inferred type for \(e_{\text{mixed}}\) is thus:

\[ e_{\text{mixed}} :: (\text{Int } \in as, \text{Bool } \in as, \text{Float } \in as) \Rightarrow \text{Exp as Float} \]

A key feature is that the constraints list explicitly the types that appear within an expression that are not visible simply by looking at the expression’s top-level type (\(\text{Exp as Float}\) in the case of \(e_{\text{mixed}}\)). As for the list \(as\) itself, it is constructed using features provided by GHC’s data type promotion (Section 2.7). That is, it is built from the types:

\[ \text{’[]} :: \forall \kappa. [\kappa] \]
\[ (;) :: \forall \kappa. \kappa \rightarrow [\kappa] \rightarrow [\kappa] \]

which arise from the promotion of the Haskell Prelude’s list data type. The quote (’) in the name of ‘[]’ is not a typographical error: it serves to distinguish it from the traditional list type constructor [ ] (of kind \(\star \rightarrow \star\)).
As we saw in Chapter 3, type-level lists can be encoded without such advanced compiler support: we could equally well use the \((\_, \_\) ) and \((\_\) ) type constructors, c.f. HL\textsc{ist} (Kiselyov et al. [33]), forming types such as \((\text{Int}, (\text{Bool}, (\_\) ))\) instead of \('\text{[Int, Bool]}\). We have opted to work with the latter set of types so as to take advantage of the rather more convenient syntax and associated kind safety.

### 4.2.1 Picking types

At this point the list of types \(\text{as}\) is purely symbolic, so if an expression \(e\) is typed as \((\tau_1 \in \text{as}, \ldots, \tau_n \in \text{as}) \Rightarrow \text{Exp as } \tau\) then the constraints state that \(\text{as}\) must contain at least the types \(\tau_1, \ldots, \tau_n\) for \(e\) to have type \(\text{Exp as } \tau\), whatever as happens to be.

Given a constrained type, one thing we should certainly be able to do is instantiate some or all of the types specified in the constraints. Thus, in the same way that we can instantiate \(a\) in the type of Haskell’s built-in function \(\text{abs :: Num } a \Rightarrow a \rightarrow a\) to the specific instance \(\text{abs :: Int} \rightarrow \text{Int}\), for example, so we should be able to do the same for constraints involving \((\in\) ). We would therefore like all of the following to be valid instantiations for the type of \(e_{\text{mixed}}\) above:

\[
\begin{align*}
e_{\text{mixed}} :: (\text{Bool} \in \text{as}, \text{Float} \in \text{as}) & \Rightarrow \text{Exp (Int : as) Float} \\
e_{\text{mixed}} :: \text{Exp } \mathcal{'}[\text{Bool}, \text{Float}, \text{Int}] \text{ Float} \\
e_{\text{mixed}} :: \text{Exp } \mathcal{'}[\text{Char}, \text{Bool}, \text{Float}, \text{Int}] \text{ Float}
\end{align*}
\]

whilst an attempted instantiation such as

\[
e_{\text{mixed}} :: \text{Exp } \mathcal{'}[\text{Bool}, \text{Int}] \text{ Float}
\]

should fail to type check.

To see how this can be achieved we now develop the implementation of \((\in\) ) and discuss some possible variations. We have a basic requirement to ensure that a constraint \(a \in \text{as}\) is only satisfied when the list of types \(\text{as}\) contains \(a\). The usual approach when dealing with lists is to assert that if the head of \(\text{as}\) is \(a\) then the constraint is satisfied trivially and, if not, to seek evidence that \(a\) appears somewhere in the tail of \(\text{as}\). The following GADT describes these two cases:

\[
\begin{align*}
data \text{ Evidence } a \text{ as } \text{where} \\
\text{Head} :: \text{ Evidence } a (a : \text{as}) \\
\text{Tail} :: (a \in \text{as}) \Rightarrow \text{ Evidence } a (b : \text{as})
\end{align*}
\]

This states that if an object of type \(\text{Evidence } a \text{ as}\) is \(\text{Head}\) then \(a\) is at the head of the \(\text{as}\), and that if it is \(\text{Tail}\) then \(a\) is in the list, but is not at the head. Here the recursive check into the tail of the list is performed by virtue of the constraint \(a \in \text{as}\) in the definition of \(\text{Tail}\), but it could also instead be made explicit, as in:
data Evidence a as where
   Head :: Evidence a (a : as)
   Tail :: !(Evidence a as) \rightarrow Evidence a (b : as)

where the strictness annotation (!(!(...))) is necessary to ensure that ‘evidence’ such as Tail ⊥ cannot be constructed. In either case, if a is not in as then it should be impossible to create an object whose type matches that of either Head or Tail. Thus, if an object has type Evidence a as then it is irrefutably the case that a is in as, for otherwise we have a type error! This is the key property that governs our definition of (∈):

class a ∈ as where
   evidence :: Evidence a as

which can be read: ‘a is an element of as and the value evidence is either Head or Tail, depending on where in the list a resides.’ There are no other possibilities. Which of the two cases we have is now determined by the instance declarations:

instance a ∈ (a : as) where
   evidence = Head

instance (a ∈ as) \Rightarrow a ∈ (b : as) where
   evidence = Tail

Thus, for example, the following:

   evidence :: Evidence Int ‘[Int, Char]
   evidence :: Evidence Int ‘[Float, Int]

have values Head and Tail respectively, whereas there is no instance of (∈) which provides evidence :: Evidence Int ‘[Bool].

Note that it is important that there is no case for ‘[] because we want there to be no evidence that a is an element of ‘[]), which is surely a lie! However, it is interesting to see what happens if we try to define such an instance. Rather conveniently, if we attempt the following:

instance a ∈ ‘[] where
   evidence = . . .

then the only thing we can put on the right-hand side is ⊥ (or error). Looking at this another way, such an instance can successfully encode the lie that a is an element of ‘[], but the body cannot provide the evidence. The refute function calls the bluff of such liars by forcing the evaluation of the given ‘evidence’:

refute :: Evidence a ‘[] \rightarrow b
refute x
   = seq x (error “Impossible“)
We note that, necessarily, the two instances of \((\in)\) shown overlap, and may thus only be used when one is recognisably more specific than the other. This means that we may only eliminate \((\in)\) constraints when reasoning about lists of ground types: there is no way in general for the compiler to know whether the type variable \(a\) is a member of the list \((b:a)\), i.e. whether \(a\) and \(b\) are the same type. This is not a serious limitation, however, as the constraints are typically only eliminated at the point where a concrete type must be picked anyway. We will return to this point in Section 5.5.

4.2.2 Platform-specific constraints

Let us once more return to the implementation of the compileSM function, whose type we shall reset to the entirely non-discriminating:

\[
\text{compileSM} :: \text{Exp as } a \to \text{String}
\]

Yet again, the AddE, EqE and CondE cases are accepted without complaint; the ValueE clause will as usual require more attention. As it stands, we have the following:

\[
\begin{align*}
\text{data} & \quad \text{Exp as } a \text{ where} \\
& \quad \text{ValueE} :: (a \in as) \Rightarrow a \to \text{Exp as } a \\
& \quad \cdots
\end{align*}
\]

\[
\begin{align*}
\text{compileSM} & :: \text{Exp as } a \to \text{String} \\
\text{compileSM} & (\text{ValueE } x) \\
& = \text{"PUSH " ++ show (toInt x) ++ \"\n\"} \\
& \cdots
\end{align*}
\]

From the definition of Exp above, all we know about the type of \(x\) (\(b\), say) is that \(b \in as\). For this particular implementation we need to establish the fact that \(b\) is also either an integer or a boolean, i.e. we need to satisfy the constraint \(\text{IntBool } b\) before we can invoke \(\text{toInt}\).

The key is to look at the type of compileSM itself. There is clearly a platform-specific requirement that all types in a given expression must be either Ints or Bools so the place to specify this constraint must be in the definition of compileSM itself:

\[
\text{compileSM} :: \text{AllIntBool as } \Rightarrow \text{Exp as } a \to \text{String}
\]

The idea is for the AllIntBool type class to define a ‘wrapper’ function, \(\text{elemToInt}\), whose role is to pick the right instance of \(\text{IntBool}\) i.e. either the \(\text{Int}\) or \(\text{Bool}\) instance; this involves convincing the type checker that such an instance exists. In order to do this \(\text{elemToInt}\) needs to carry with it the list of constrained types \(as\):

\[
\begin{align*}
\text{class} & \quad \text{AllIntBool as where} \\
& \quad \text{elemToInt} :: (b \in as) \Rightarrow \text{Proxy as } \to b \to \text{Int}
\end{align*}
\]
What is the role of the Proxy type? elemToInt possesses a context that states that \( b \in \text{as} \). To be able to call elemToInt we will need to pass an argument that makes the type \text{as} concrete, such that the Haskell compiler can pick the correct instance of AllIntBool. Usually this would involve passing an argument of type \text{as}, but \text{as} has kind \([\star]\) whereas types of function arguments must have kind \(*\). The Proxy data type suffices to effect the conversion:

```haskell
data Proxy as = Proxy
```

This fixes the type at compile-time at the expense of passing additional values at run-time. The same effect could be achieved by attributing the phantom type \text{as} to elemToInt’s result, as in:

```haskell
newtype WithList as a = WithList \{ withoutList :: a \}
class AllIntBool as where
  elemToInt :: (b \(\in\) \text{as}) \(\Rightarrow\) b \(\rightarrow\) WithList as \text{Int}
```

This has the advantage of introducing no run-time overhead, but requires us to manually pattern-match off the WithList constructor from elemToInt’s result, as in:

```haskell
compileSM (ValueE x)
  = "PUSH" ++ show ((withoutList :: WithList as a \rightarrow a) (elemToInt x)) ++ "\n"
```

where the type variable \text{as} in withoutList’s definition is scoped by the type signature of compileSM. We believe, however, that the passing of proxy values makes for a cleaner presentation. Assuming the Proxy-based solution, it will be useful to complete the compiler at this point in order to see how the additional argument comes into play:

```haskell
compileSM (ValueE x)
  = "PUSH " ++ show (elemToInt (Proxy :: Proxy as) x) ++ "\n"
```

In each case we note that lexically-scoped type variables (Section 2.5) are crucial in calling elemToInt at the correct type. Let’s now build the AllIntBool class instances, beginning with the instance for \((:)\). If \text{a} is an instance of IntBool and the list \text{as} is AllIntBool then we need to extend the constraint to the list \((\text{a} : \text{as})\):

```haskell
instance (IntBool a, AllIntBool as)
  \Rightarrow AllIntBool (a : as) where
  elemToInt _ x = 

The trick here is to observe the type of elemToInt:

```haskell
elemToInt :: (IntBool a, AllIntBool as, \text{b} \(\in\) (a : as))
  \Rightarrow Proxy (a : as) \rightarrow b \rightarrow \text{Int}
```
Here, both $a$ and $b$ are type variables so in the event that $b$ is at the head of the list ($a : as$), we shall have that $a \equiv b$ and hence IntBool $b$. In this case, we can therefore apply toInt to $x$ and we are done. If $b$ is in the tail of the list ($a : as$) then we require a proof that $b \in as$, in which case we shall proceed by recursion. How do we know whether $b$ is at the head of $as$? We use the evidence method of the corresponding ($\in$) instance:

\begin{verbatim}
instance (IntBool a, AllIntBool as) ⇒ AllIntBool (a : as) where
elemToInt _ (x :: b) = case evidence :: Evidence b (a : as) of
  Head → toInt x
  Tail → elemToInt (Proxy :: Proxy as) x
\end{verbatim}

Notice that in the recursive call to elemToInt we must reconstruct a new proxy object, as its type must reflect that of the tail of ($a : as$). This proxy argument is never referred to at the object level; its role is simply to pick the required type instance for AllIntBool.

An instance for $’[]$ is needed because the AllIntBool instance for $a : ’[]$ (see above) requires the two constraints IntBool $a$ and AllIntBool $’[]$. Of course, in this case it must be the case that the corresponding evidence is Head for otherwise the type we are looking for would not be in the list; a recursive call to elemToInt with argument (Proxy :: Proxy $’[]$) can therefore never occur. We thus need the $’[]$ instance, but not its associated implementation of elemToInt:

\begin{verbatim}
instance AllIntBool ’[] where
elemToInt _ (x :: b) = refute (evidence :: Evidence b ’[])
\end{verbatim}

Note that we could simply have made the right-hand side $\bot$ but we instead ‘call the bluff’ of any instance of the form $b \in ’[]$ using the refute function defined earlier. We shall return to this point in Section 4.6, in which we will encode our solution in Agda. In doing so we gain the ability to define the equivalent of the $’[]$ instance without having to provide an absurd definition of elemToInt.

### 4.2.3 Implementation cost

The process of picking concrete types for each ($\in$) constraint (Section 4.2.1) incurs a compile-time overhead, as we must ensure that each ($\in$) constraint is satisfied by the given list of concrete types. For example, for the DSL expression $e$ with type ($\tau_1 \in as, \ldots, \tau_n \in as$) $⇒$ Exp $as$ $a$ we can impose any type of the form $e ::$ Exp $\tau s$ $a$ provided each $\tau_i$ appears in $\tau s$. If $\tau s$ contains $m$ types then the cost of the static type check is $O(mn)$.

There is also a run-time overhead, however, which depends on how the type classes involved are implemented. In what follows we shall assume the use of the dictionary-passing transformation of Wadler and Blott [62] (as mentioned in Section 2.4) to explain in principle what must
happen at run-time. Each overloaded function used in a given implementation (e.g. \texttt{toInt} in compileSM) must be invoked at the correct type. This is not done by traversing a list of types, however, as no type information is retained at run-time. Instead, each constraint in a function’s type (e.g. \( C \Rightarrow \ldots \)) is translated by the Haskell compiler into an additional argument (\( C \rightarrow \ldots \)) that implements a dictionary of functions that correspond to a specific instance of the corresponding class (here \( C \)). Figure 4.1 shows this in the context of our example above, where the invocation of the functions elemToInt, evidence and the pattern-match on Tail at different types result in the introduction of different dictionaries at run-time.

When the call evidence \( de \) in consInstance returns Head, the knowledge that the types \( a \) and \( b \) are equal means that the toInt method of the \( di \) dictionary may be invoked, producing the necessary result. In the case that evidence \( de \) returns Tail \( de' \), the proof that the type \( b \) is in the tail \( as \) is extracted and passed to a recursive invocation of elemToInt. Observe then that the cost of invoking toInt at type \( \tau \) in the example above is linearly proportional to the index, \( n \) say, of \( \tau \) in the list \( \tau s \) in the imposed type \( \text{Exp}\ \tau s\ a \). In short, we do not end up ‘searching’ for the correct dictionary for \( \tau \) at run-time; instead elemToInt is invoked exactly \( n \) times whereupon its additional dictionary argument provides the required \( \tau \) instance of the \text{IntBool} class.

4.2.4 Type-safe heterogeneity

Let us now consider what happens if we wish to apply several constrained functions to a single DSL expression. We begin by reworking the type of the pretty function from Section 4.1 in order to mirror that of compileSM:

\[
pretty :: \text{AllShowable}\ as \Rightarrow \text{Exp}\ as\ a \rightarrow \text{String}
\]

As one might expect, the constraint \text{AllShowable}\ as provides a guarantee that every type in \( as \) has a textual representation. Now consider the previous example, where a DSL expression is to be used in two different contexts:

\[
f\ e = \ldots (\text{compileSM}\ e) \ldots (\text{pretty}\ e) \ldots
\]

For this to type check, \( e \) must be an expression of type \( \text{Exp}\ as\ a \). The use of compileSM requires the constraint \text{AllIntBool} as to be satisfied. Additionally, the use of pretty introduces the constraint \text{AllShowable} as. Assuming that no other context is required, \( f \)'s type will thus be inferred as:

\[
f :: (\text{AllIntBool}\ as, \text{AllShowable} as) \Rightarrow \text{Exp}\ as\ a \rightarrow \ldots
\]

Note that \( e \)'s type, \( \text{Exp}\ as\ a \), makes no reference to the constraints of either compileSM or pretty – it is completely removed from any implementation. A corollary of this is that \( e \)'s type is the same at the call sites of both functions. If it were not, \( e \) would have to be defined as a polymorphic argument: \( f \) would require a higher-rank type and the problems of Section 4.1 would reappear. This is not to say that higher-rank types cannot be used in conjunction with generic constraints, however; the particulars will be discussed in detail in Section 4.4.
data Evidence a as where
   Head :: Evidence a (a : as)
   Tail :: (a ∈ as) → Evidence a (b : as)
data a ∈ as
   = Elem { evidence :: Evidence a as }
headInstance :: a ∈ (a : as)
headInstance
   = Elem { evidence = Head }
tailInstance :: (a ∈ as) → (a ∈ (b : as))
tailInstance de
   = Elem { evidence = Tail de }
data IntBool a
   = IntBool { toInt :: a → Int }
data AllIntBool as
   = AllIntBool { elemToInt :: ∀b. (b ∈ as) → b → Int }
nilInstance :: AllIntBool '[]
nilInstance
   = AllIntBool { elemToInt = λ (de :: b ∈ '[]) (x :: b) → seq (evidence de) ⊥ }
consInstance :: IntBool a → AllIntBool as → AllIntBool (a : as)
consInstance di da
   = AllIntBool { elemToInt
       = λ (de :: b ∈ (a : as)) (x :: b) →
       \case evidence de of
           Head → toInt di x
           Tail de’ → elemToInt da de’ x
       }

Figure 4.1: The result of applying the dictionary-passing transformation [62] to the (∈), IntBool and AllIntBool type classes.
4.3 Higher-order constraints

While AllIntBool is a useful type class, its implementation is both non-trivial and tied to the needs of the compileSM function. Far better would be to write one type class which encapsulates the notion of traversing type-level lists and subsequently parameterise it by a particular platform-dependent constraint. Thanks to GHC’s recently-added support for constraint kinds (Section 2.7) we can do just that:

\[
\text{class All } c \text{ as where}
\]

All's first parameter, \( c \), is a constraint constructor. As an example, the idea is that instantiating \( c \) to be \( \text{IntBool} \) will recreate the definition of AllIntBool above. Of course, this will only be the case if \( \text{elemToInt} \) is generalised appropriately. To this end, let’s attempt to define a function \( \text{withElem} \), say, that abstracts over the result type (\( \text{Int} \), in the case of \( \text{elemToInt} \) above) , which must now carry the constraint \( c \ b \):

\[
\text{class All } c \text{ as where}
\]

\[
\text{withElem :: } (b \in as) \Rightarrow \text{Proxy } as \rightarrow (c \ b \Rightarrow d) \rightarrow d
\]

Unfortunately, while this is a valid type in the eyes of the type checker,\(^1\) this leads to ambiguous type constraints because of the fact that constraints in Haskell 'float' to the leftmost position in a type. As an example, consider the \( \text{toInt} \) function above. Its type is \( \text{IntBool } b \Rightarrow b \rightarrow \text{Int} \), a seemingly perfect fit for the above, which we would like to lead to the type:

\[
\text{withElem :: } (b \in as) \Rightarrow \text{Proxy } as \rightarrow (\text{IntBool } b \Rightarrow b \rightarrow \text{Int}) \rightarrow b \rightarrow \text{Int}
\]

However, the \( \text{IntBool} \) constraint is instead floated out to yield:

\[
(b \in as, \text{IntBool } b) \Rightarrow \text{Proxy } as \rightarrow (c \ b \Rightarrow b \rightarrow \text{Int}) \rightarrow (b \rightarrow \text{Int})
\]

which leaves an ambiguous constraint \( c \ b \) which the compiler is unable to resolve. To mitigate this issue, we transform the constraint \( c \ b \) into a data type which realises the relationship between \( c \) and \( b \):

\[
\text{data Trap } c \ b \text{ where}
\]

\[
\text{Trap :: } c \ b \Rightarrow \text{Trap } c \ b
\]

The name 'Trap' reflects the fact that this GADT traps a constraint in a value, preventing the floating described above. However, the type is in fact a dictionary: pattern-matching on a constructor \( \text{Trap} \) of type \( \text{Trap } c \ b \) will bring into scope the constraint \( c \ b \). Additionally, one may not create a value of type \( \text{Trap } c \ b \) unless the constraint \( c \ b \) is satisfiable. We arrive at:

\(^1\)Provided that higher-rank types are permitted.
class All c as where
  withElem :: (b ∈ as) ⇒ Proxy as → (Trap c b → d) → d

The instances of All are now simple generalisations of the AllIntBool class above:

instance All c' [] where
  withElem _ (f :: Trap c b → d)
      = seq (evidence :: Evidence b '[]) ⊥

instance (c a, All c as) ⇒ All c (a : as) where
  withElem _ (f :: Trap c b → d)
      = case evidence :: Evidence b (a : as) of
          Head → f Trap
          Tail → withElem (Proxy :: Proxy as) f

Note that the second instance requires Haskell’s support for *undecidable instances* (Section 2.1) as the type system cannot decide whether the constraint expressions c a and All c (a : as) can ever be the same, in which case the type checker will loop. In this case it is easy to see that no instantiation of c can ever satisfy this property, not least because c a makes no reference to as.

To illustrate the use of withElem let’s see how the compileSM function above can be defined in terms of an All c as constraint. In this case, withElem’s second argument is a function of type Trap IntBool b → String and its job is to show the integer representation of a given b. The constraint that b is an instance of IntBool is now captured by a Trap value of type Trap IntBool b which must be named explicitly in a type signature:

```
compileSM :: All IntBool as ⇒ Exp as a → String
compileSM (ValueE x)
      = "PUSH " ++ withElem (Proxy :: Proxy as) (showInt x) ++ "\n"
    where
        showInt :: b → Trap IntBool b → String
        showInt x Trap
            = show (toInt x)
```

The other two cases are unchanged. Importantly the showInt function must be explicitly typed (as shown) for otherwise the type would be inferred as:

```
showInt :: IntBool b ⇒ b → Trap c b → String
```

where the dictionary has an unconstrained (polymorphic) type. We must instead enforce the IntBool constraint on the dictionary itself. In short, we want a specific instance of showInt’s principal type that enforces the constraint relationship between IntBool and b.
4.4 Tagless representations

At this point we consider the implications, if any, of choosing GADTs as the basis for our DSL embedding. Section 3.2 demonstrated how tagless encodings often necessitate the introduction of hitherto bothersome features such as higher-rank types, for example. In this vein, let a tagless counterpart of the generically-constrained Exp GADT be defined as:

```haskell
class TaglessExp e where
    valueE :: (a ∈ as) ⇒ a → e as a
    addE :: (Num a, a ∈ as)
           ⇒ e as a → e as a → e as a
    eqE :: (Eq a, a ∈ as, Bool ∈ as)
         ⇒ e as a → e as a → e as Bool
    condE :: (a ∈ as, Bool ∈ as)
           ⇒ e as Bool → e as a → e as a
           → e as a
```

The compileSM function must be recast in terms of a data type and a corresponding TaglessExp instance that defines the valueE, addE, eqE and condE functions. The data type is straightforward and may make use of All:

```haskell
newtype CompileSM as a = CompileSM (All Int Bool as ⇒ String)
```

There is a small complication with the definition of valueE, which can be seen when we try to define the TaglessExp instance:

```haskell
instance TaglessExp CompileSM where
    valueE x = ...
```

At this point we need to construct a Proxy of type Proxy as, but we have no as to refer to. The solution is to define a helper function valueE’, whose type signature imposes the constraint a ∈ as on x’s type:

```haskell
valueE’ :: (a ∈ as) ⇒ a → CompileSM as a
valueE’ x
    = CompileSM
        (“PUSH " ++ withElem (Proxy :: Proxy as) (showInt x) ++ “\n”)
```

showInt is as defined earlier. We can now complete the instance:

```haskell
instance TaglessExp CompileSM where
    valueE
```
\[
= \text{valueE'}
\]
\[
\text{addE} \ (\text{CompileSM} \ s_1) \ (\text{CompileSM} \ s_2)
= \text{CompileSM} \ (s_1 + s_2 + "ADD\n")
\]
\[
\text{eqE} \ (\text{CompileSM} \ s_1) \ (\text{CompileSM} \ s_2)
= \text{CompileSM} \ (s_1 + s_2 + "EQ\n")
\]
\[
\text{condE} \ (\text{CompileSM} \ p) \ (\text{CompileSM} \ t) \ (\text{CompileSM} \ f)
= \text{CompileSM} \$
\]
\[
p + "\text{CMP } #0\n" + "\text{BEQ L1}\n" + t +
"\text{BR L2}\n" + "L1: " + f + "L2 :"
\]

The familiar compileSM function is now built simply by picking the correct instance of the TaglessExp class:

\[
\text{compileSM} :: \text{All \ Int\Bool \ as} \Rightarrow \text{CompileSM \ as \ a} \rightarrow \text{String}
\]
\[
\text{compileSM} \ (\text{CompileSM} \ s)
= s
\]

So, what of heterogeneity? In contrast with GADT representations, tagless encodings are by construction parameterised by the implementation they target i.e. \( e \). Consequently, invoking multiple implementations of a single expression is not possible with the definitions given so far. Assuming the presence of Pretty, a tagless remodelling of the pretty-printing pretty function from Section 4.2.4, we may revisit our earlier example:

\[
f \ e = \ldots (\text{compileSM} \ e) \ldots (\text{pretty} \ e) \ldots
\]

Naturally, \( f \) will not type check because the type variable \( e \) cannot be instantiated to both CompileSM and Pretty. Rather than assigning \( f \) a rank-2 type, let us close our terms polymorphically once and for all:

\[
\textbf{newtype} \ \text{AnyTaglessExp \ as} \ a
= \text{AnyTaglessExp} \ (\forall e. \ \text{TaglessExp} \ e \Rightarrow e \ as \ a)
\]

Perhaps surprisingly, \( f \ can \ be \ rewritten \ to \ accept \ a \ value \ of \ type \ \text{AnyTaglessExp}:

\[
f \ (\text{AnyTaglessExp} \ e) = \ldots (\text{compileSM} \ e) \ldots (\text{pretty} \ e) \ldots
\]

The key point to note is that the type \( as \ 'escapes' \ the \ rank-2 \ type \ of \ AnyTaglessExp, \ meaning \ that \ it \ can \ be \ constrained \ in \ the \ desired \ manner. \ This \ is \ not \ true \ of \ \( e \), \ which \ makes \ no \ appearance \ in \ a \ type \ such \ as \ AnyTaglessExp \ as \ a \ and \ consequently \ cannot \ be \ constrained \ outside \ the \ definition \ of \ the \ \text{AnyTaglessExp} \ constructor. \ However, \ despite \ representing \ an \ expression's \ target \ platform, \ the \ type \ \( e \ plays \ no \ role \ in \ platform-specific \ constraints, \ and \ may \ therefore \ be \ hidden \ safely. \ In \ effect, \ \text{AnyTaglessExp} \ is \ equivalent \ to \ the \ \text{Exp} \ GADT \ introduced \ earlier.
4.5 Exploiting kind polymorphism to constrain operations

The previous sections have demonstrated how we may restrict the types of values that are introduced into a computation. However, the \((\in)\) and All type classes in fact provide a more general framework for expressing properties of data types. As an example, we shall see now that we may also bound the operations that are used in an expression, thanks in part to kind polymorphism (Section 2.7). While as of now the \((\in)\) and All type classes have been used only to constrain lists of kind \([\ast]\), their definitions afford them the following more general kinds:

\[
(\in) :: \forall \kappa. \kappa \rightarrow [\kappa] \rightarrow \text{Constraint} \\
\text{All} :: \forall \kappa. (\kappa \rightarrow \text{Constraint}) \rightarrow [\kappa] \rightarrow \text{Constraint}
\]

Here, as one might expect, \(\kappa\) may be instantiated to any kind. What does this buy? Consider extending the Exp GADT to record the operations, \(os\) say, in addition to the types, that are used in the construction of an expression:

```haskell
data Exp as os a where
  ...
```

We would like to use the \((\in)\) type class to constrain \(os\), similar to the manner in which we constrained types using \(as\). We can achieve this by promoting the constructors of a data type such as:

```haskell
data Op = AddO | EqO | CondO
```

to the type level. The modifications required to Exp are straightforward:

```haskell
data Exp as os a where
  ValueE :: (a \in as) \Rightarrow a \rightarrow Exp as as a
  AddE :: (Num a, a \in as, AddO \in os)
        \Rightarrow Exp as as a \rightarrow Exp as as a \rightarrow Exp as as a
  EqE :: (Eq a, a \in as, Bool \in as, EqO \in os)
       \Rightarrow Exp as as a \rightarrow Exp as as a \rightarrow Exp as as a
  CondE :: (a \in as, Bool \in as, CondO \in os)
         \Rightarrow Exp as as Bool \rightarrow Exp as as a \rightarrow Exp as as a
         \Rightarrow Exp as as a
```

Recall that GHC will not promote GADTs due to limitations in its implementation (Section 2.7); ideally we’d like to be able to promote the constructors of the Exp type itself, as in:

```haskell
data Exp as os a where
  ...
```
AddE :: (Num a, a ∈ as, AddE ∈ os) ⇒ Exp as os a → Exp as os a → Exp as os a

for example. Irrespective of this, the modified version of Exp above allows us to choose which operations are permitted in a given implementation. As an example, suppose we wish to generate code for an architecture in which conditional branching is undesirable (Nvidia’s CUDA platform, for example). Once again, a type class may be defined to capture the operations that are supported:

```haskell
class CUDACompatible o
instance CUDACompatible AddO
instance CUDACompatible EqO
```

The extension of CUDACompatible over a list os is then handled by the All class (for brevity we do not detail a complete CUDA compiler):

```haskell
compileCUDA :: All CUDACompatible os ⇒ Exp as os a → String

compileCUDA (ValueE x) = ... 
compileCUDA (AddE e1 e2) = ... 
compileCUDA (EqE e1 e2) = ...
```

We omit a case which pattern-matches CondE as compileCUDA’s type guarantees that its argument will not contain a CondE term. The presence of such a term would introduce the context CondO ∈ os. Since compileCUDA’s type mentions the constraint All CUDACompatible os, the instance CUDACompatible CondO would subsequently be required. By design this instance doesn’t exist and so the application of compileCUDA to such a term would be ill-typed. GHC is not capable of such reasoning however and so adding a CondE clause to the definition of compileCUDA is well-typed. In Section 7.9 we explore how an alternative (equivalent) encoding allows us to write a provably total Agda program in which adding, for example, a CondE case is ill-typed.

Total or not, parameterising Exp by both the types it contains (as) and the operations it uses (os) seems a little cumbersome. A neater approach is to use some form of union operation, but at the type level. For example, we can use a promoted version of Haskell’s Either type:

```haskell
data Either a b = Left a | Right b
```

With this, Exp can instead be parameterised by a single list, ts, of kind [Either ⋆ Op]; each item of ts will be either a type or an operation:

```haskell
data Exp ts a where 
ValueE :: (Left a ∈ ts) ⇒ a → Exp ts a
```
AddE :: (Num a, Left a ∈ ts, Right AddO ∈ ts) ⇒ Exp ts a → Exp ts a → Exp ts a
EqE :: (Eq a, Left a ∈ ts, Left Bool ∈ ts, Right EqO ∈ ts) ⇒ Exp ts a → Exp ts a → Exp ts Bool
CondE :: (Left a ∈ ts, Left Bool ∈ ts, Right CondO ∈ ts) ⇒ Exp ts Bool → Exp ts a → Exp ts a → Exp ts a

However, the All family of type classes introduced in Section 4.3 doesn’t fit well with this because All constrains all items in a list, whereas we need to be able to constrain either the as (Left) or the os (Right). We must therefore adapt the class by decomposing it into a pair of classes, each designed to constrain one of the types present in the list:

class AllLeft c ts where
  withLeftElem :: (Left a ∈ ts) ⇒ Proxy ts → (Trap c a → d) → d

class AllRight c ts where
  withRightElem :: (Right b ∈ ts) ⇒ Proxy ts → (Trap c b → d) → d

If we consider the AllLeft class (the workings of AllRight follow suit), we see that we now need two instances for (:) that will begin:

instance (c a, AllLeft c ts)
  ⇒ AllLeft c (Left a : ts) where
    ...

instance AllLeft c ts ⇒ AllLeft c (Right a : ts) where
  ...

In the first case a type Left a can only be added to the list ts if a satisfies the constraint c. The instance is thus similar to that given in the definition of All:

instance (c a, AllLeft c ts)
  ⇒ AllLeft c (Left a : ts) where
    withLeftElem _ (f :: Trap c b → d) = case evidence :: Evidence (Left b) (Left a : ts) of
      Head → f Trap
      Tail → withLeftElem (Proxy :: Proxy ts) f

As for the second instance, it is in fact simpler - types of the form Right a may be added to the list ts regardless:

instance AllLeft c ts ⇒ AllLeft c (Right a : ts) where
  withLeftElem _ (f :: Trap c b → d)
class All tag c ts where

  withTaggedElem :: (tag a ∈ ts)
  ⇒ Proxy tag → Proxy ts → (Trap c a → b) → b

instance (c a, All Left c ts) ⇒ All Left c (Left a : ts) where

  withTaggedElem p _ (f :: Trap c b → d)
  = case evidence :: Evidence (Left b) (Left a : ts) of
    Head → f Trap
    Tail → withTaggedElem p (Proxy :: Proxy ts) f

...
4.6 Proving the totality of generically constrained functions

Recall the compileCUDA function of Section 4.5:

\[
\text{compileCUDA :: All CUDACompatible os } \Rightarrow \text{Exp as os a } \rightarrow \text{String}
\]

\[
\begin{align*}
\text{compileCUDA } (\text{ValueE } x) &= \ldots \\
\text{compileCUDA } (\text{AddE } e_1 e_2) &= \ldots \\
\text{compileCUDA } (\text{EqE } e_1 e_2) &= \ldots
\end{align*}
\]

Despite the complete omission of a clause matching the CondE constructor, it was argued that this function is total: there is no instance CUDACompatible CondO and so adding such a case would cause the function to be ill-typed. However, since type classes are open, GHC cannot determine whether compileCUDA is total or not because an instance CUDACompatible CondO may be added at a later point. In fact, GHC’s totality checker is too conservative to agree with us even when presented with a closed encoding of the same program (a point which will be elaborated later on). In this section we will define an encoding of programs written with using our technique in Agda (Section 3.8). The objective is to transform any well-typed Haskell program that is in fact total into a well-typed Agda program that is, by construction, provably total. In doing so we will rely on Agda’s notion of absurdity:

\[
\text{data Fin : Nat } \rightarrow \text{Set where}
\]

\[
\begin{align*}
\text{ZeroF : } &\{n : \text{Nat}\} \rightarrow \text{Fin (Suc n)} \\
\text{SucF : } &\{n : \text{Nat}\} \rightarrow \text{Fin n } \rightarrow \text{Fin (Suc n)}
\end{align*}
\]

\[
f : \{A : \text{Set}\} \rightarrow \text{Fin Zero } \rightarrow \text{A}
\]

\[
f ()
\]

Fin \(n\) is a type which embodies the set of natural numbers small than \(n\). The type Fin Zero is thus empty, as there are no natural numbers smaller than zero. As a result there are no constructors which may be pattern-matched by the function \(f\). We thus write the absurd pattern () and omit a right-hand side. We cannot simply omit the whole case as the expression \(f x = f x\) is type correct and could hence be provided (though it would be rejected by Agda’s termination checker). In essence, the aim of this section is to define an encoding under which the compileCUDA function’s CondE clause (or equivalent thereof) is absurd.

4.6.1 Stratification and universe polymorphism

In Section 3.8, the type of length-indexed vectors was presented in Agda as:

\[
\text{data Vec \( (A : \text{Set}) : \text{Nat } \rightarrow \text{Set where}}
\]

\[
\begin{align*}
[] &\text{ : Vec A Zero}
\end{align*}
\]
\[ n :: \{ n : \text{Nat} \} \rightarrow A \rightarrow \text{Vec } A \ n \rightarrow \text{Vec } A \ (\text{Suc } n) \]

Similarly, the simpler notion of a list may be introduced as:

```agda
data List (A : Set) : Set where
  [] : List A
  _::_ : A \rightarrow List A \rightarrow List A
```

Regrettably, neither of these definitions is suitable for representing a list of *types*. It is true that a list of values may be used *at the type level*, viz.:

```agda
data _≡_ {A : Set} (x : A) : A \rightarrow Set where
  Refl : x ≡ x
  p : [] ≡ []
  p
  = Refl
```

where \( p \)'s type states that \( p \) is a *proof* that the empty list is equal to itself (the justification being reflexivity). However, the term:

```agda
Nat :: []
```

is ill-typed. The reason is one of *stratification*. As its definition states, \( \text{Nat} \) itself has type \( \text{Set} \):

```agda
data Nat : Set where

```

Following suit, the List type constructor accepts a type (such as \( \text{Nat} \)) of type \( \text{Set} \) as its parameter:

```agda
data List (A : Set) : Set where

```

The list itself then contains values of type \( A \). If the values are themselves to be types, then it would seem that \( A \) itself must be \( \text{Set} \). Herein lies the issue: a typing rule \( \text{Set} : \text{Set} \) would make the type system inconsistent (allowing us to encode, for example, Hurken's paradox [\text{26}]) and is accordingly not defined. Agda's solution is to assign \( \text{Set} \) the type \( \text{Set}^1 \), where there is an infinite hierarchy of stratified universes \( \text{Set}, \text{Set}^1, \text{Set}^2, \ldots \) such that \( \text{Set}^n : \text{Set}^{(n+1)} \).

Data types which are independent of the position at which they reside in the \( \text{Set} \) hierarchy may be written in Agda using the language's support for *universe polymorphism*. The humble list type becomes:

```agda
data List {i : Level} \{ A : \text{Set}_i \} : \text{Set}_i where
  [] : List A
  _::_ : A \rightarrow List A \rightarrow List A
```

73
List is now parameterised by the level $i$ (of type Level) which may be used as an argument to the general-purpose type constructor Set. The type now functions as we desire: the type List Nat has type Set and possesses inhabitants such as $[]$ and Zero :: $[]$; the type List Set has type Set$_i$ and is inhabited by values such as Bool :: Nat :: $[]$.

4.6.2 Type classes as universes

Given the ability to define lists of types, the next step is to encode membership – the ($\in$) class and the Evidence data type. Since Agda lacks type classes and only possesses the limited notion of implicit and 'instance' arguments (Devriese and Piessens [18]), we will encode type classes as universes (Section 3.8). Importantly, this transformation will result in a set of closed types. The universe corresponding to the ($\in$) class is in fact the Evidence type:

```agda
data _\in_ \{ i \} \{ T : \text{Set}_i \} (A : T) : \text{List} T \rightarrow \text{Set}_i \text{ where}
Head : \{ AS : \text{List} T \} \rightarrow A \in (A :: AS)
Tail : \{ AS : \text{List} T \} \{ B : T \} \rightarrow A \in AS \rightarrow A \in (B :: AS)
```

For comparison, the equivalent kind-annotated Haskell GADT would be:

```haskell
data (a :: $\kappa$) $\in$ (as :: $[\kappa]$) :: $\star$ where
Head :: a $\in$ (a :: as)
Tail :: a $\in$ as $\rightarrow$ a $\in$ (b :: as)
```

The primary differences are the implicit type arguments included in the Agda version, though Agda does provide a $\forall$-based shorthand for just this purpose:

```agda
data _\in_ \{ i \} \{ T : \text{Set}_i \} (A : T) : \text{List} T \rightarrow \text{Set}_i \text{ where}
Head : \forall\{ AS \} \rightarrow A \in (A :: AS)
Tail : \forall\{ AS B \} \rightarrow A \in AS \rightarrow A \in (B :: AS)
```

Now the types of the variables AS and B will be inferred by the type checker. For the sake of generalisation we have again made use of universe polymorphism; note that $i$'s type need not be given since its use as an argument to Set reveals that it is a Level.

Figure 4.3 illustrates how a Haskell type $\mathcal{E}$ may now be rewritten to use the $\in \in$ type constructor. Here we have opted for the more general definition in which a type is parameterised by a single list of items which are either types or operations, where each operation notionally describes a constructor. The ($\lor$) type constructor is thus a more fashionably-named (and exotically-typed) version of Haskell’s Either:

```agda
data _\lor_ \{ i j \} \{ A : \text{Set}_i \} \{ B : \text{Set}_j \} : \text{Set}_{(i \sqcup j)} \text{ where}
Left : A \rightarrow A \lor B
Right : B \rightarrow A \lor B
```

4. Level itself is a small type and has type Set.
The function \( \texttt{⊔} \) : \texttt{Level} \(→\) \texttt{Level} \(→\) \texttt{Level} returns the maximum of two levels. The \( \texttt{(\lor)} \) type constructor may thus be used to construct the union of two types at different levels in the universe hierarchy, resulting in a type which resides at whichever level is higher. This is necessary if, in the case of our CUDA compiler, for example, we are to have \texttt{Op} mirror its Haskell counterpart and be declared as a small type (of type \texttt{Set}):

\[
\textbf{data} \ \texttt{Op} : \texttt{Set} \ \textbf{where} \\
\text{Value} : \texttt{Op} \\
\text{Add} : \texttt{Op} \\
\text{Eq} : \texttt{Op} \\
\text{Cond} : \texttt{Op}
\]

Of course, there is nothing stopping us from requiring that \texttt{Op} have type \texttt{Set}_1 (i.e. that of \texttt{Set}), but this is arguably cheating: a type should be defined at the lowest level at which it makes sense to do so. We shall thus stick with the more complicated definition of \( \texttt{(\lor)} \).

Since the purpose of our proof is totality with relation to the operations handled by a function, we shall concentrate on constraining only the Right elements of a list – constraining the Left elements follows similarly. The universe of the \texttt{AllRight} class and its three instances thus follows:

\[
\textbf{data} \ \texttt{AllRight} \ \{ \ i \ j \} \ \{ \ S : \texttt{Set}_i \} \ \{ \ T : \texttt{Set}_j \} \ \{ \ C : T \rightarrow \texttt{Set}_j \} \ \textbf{where} \\
\text{None} : \texttt{AllRight} \ C \ []
\]

Figure 4.3: Encoding a Haskell type \( \mathcal{E} \) and its \((\in)\) constructors in Agda.
Left : ∀{A TS} → AllRight C TS → AllRight C (Left A :: TS)
Right_∧_ : ∀{B TS} → C B → AllRight C TS → AllRight C (Right B :: TS)

In the absence of constraints AllRight is instead parameterised by an arbitrary function C. The constructors None and Left are straightforward; the former corresponds to a proof that an empty list vacuously contains a set of Right-wrapped elements to which C may be applied, while the latter allows a such a proof to be extended indefinitely with irrelevant Left-wrapped elements. The Right_∧_ constructor extends a proof with a Right-wrapped element by accepting a proof that the function C may be applied to such an element – the value C B.

As for the member function withRightElem, it is the decoder for the AllRight universe, collecting the once separate instances into a single implementation:

```
withRightElem : { i j : Level} { S : Set i} { T : Set j}
           { TS : List (S ∨ T)} { B : T} { D : Set j} { C : T → Set j}
            → AllRight C TS → Right B ∈ TS → (C B → D) → D
withRightElem None () f
withRightElem (Left ds) (Tail e) f
  = withRightElem ds e f
withRightElem (Right d ∧ ds) Head f
  = f d
withRightElem (Right d ∧ ds) (Tail e) f
  = withRightElem ds e f
```

 Particularly pleasing is the fact that the clause matching None is understood by Agda to be absurd: recall in Section 4.2.2 that the need to provide a function body prompted the unsatisfactory solution of calling the bluff of a lying user. Here we may simply omit such a right-hand side, safe in the knowledge that it will never be reached.

All that remains is to encode each class S describing a set of supported operations as a data type, as in Figure 4.4. Each instance of the form S o forms a constructor o : S o, with unsupported operations possessing no equivalent constructor. Encoding the CUDACompatible class, for instance, we arrive at:

```
data CUDACompatible : Op → Set where
  Value     : CUDACompatible Value
  Add       : CUDACompatible Add
  Eq        : CUDACompatible Eq
```

Key is the absence of a constructor Cond : CUDACompatible Cond. With this, we may at last write a total function. Given a Haskell function:

```
f :: ∀vs. AllRight S ts ⇒ E ts α → ω
f ( O p_1 p_2 ... p_n)
```
class \( S \ o \ \text{where} \ldots \\
\text{instance} \ S \ o_1 \ \text{where} \ldots \\
\text{instance} \ S \ o_2 \ \text{where} \ldots \\
\ldots \\
\text{instance} \ S \ o_n \ \text{where} \ldots \\

\text{data} \ S : \text{Op} \to \text{Set} \ \text{where} \\
o_1 : S \ o_1 \\
o_2 : S \ o_2 \\
\ldots \\
o_n : S \ o_n

Figure 4.4: Encoding a class \( S \) of supported operations as a closed Agda data type.

\[
\begin{align*}
&= e \\
&\ldots
\end{align*}
\]

construct an equivalent Agda function as follows:

\[
\begin{align*}
f : \forall \{\text{vs}\} \to \text{AllRight} \ S \ \text{vs} \to \mathcal{E} \ \text{vs} \ \alpha \to \omega \\
f \ da \ (O \ \alpha_1 \ldots \alpha_m \ o \ p_1 \ p_2 \ldots \ p_n) \\
&= e \\
&\ldots
\end{align*}
\]

Furthermore, for each constructor operation \( O_x \) for which there is no \( S \) instance, i.e. for each unsupported operation, append to \( f \) a clause of the form:

\[
\begin{align*}
f \ da \ (O_x \ \alpha_1 \ldots \alpha_m \ o \ldots) \ \textbf{with} \ \text{withRightElem} \ da \ o \ \text{id} \\
&\ldots \ | \ ()
\end{align*}
\]

Figure 4.5 performs this transformation on the compileCUDA function, producing the desired provably-total compiler. The additional argument \( da \) results from the reification of the Haskell AllRight constraint as an explicit argument and is a dictionary (or proof) which states that every Right element of the list of types \( \text{vs} \) possesses a constructor in \( S \).

For the unsupported operations, the trick is the use of a \textbf{with} clause, which permits us additional dependent pattern-matches, each of which may provide additional information (analogous to the constraints introduced by pattern-matching on GADT constructors discussed in Section 2.3). Here we delegate to \text{withRightElem} which, importantly, is given a universe-polymorphic identity function:
compileCUDA : ∀{A TS} → AllRight CUDACompatible TS → Exp TS A → Bool
compileCUDA da (Value α o x)
    = True
compileCUDA da (Add α₁ α₂ o e₁ e₂)
    = True
compileCUDA da (Eq α₁ α₂ o e₁ e₂)
    = True
compileCUDA da (Cond α₁ α₂ o p t f) with withRightElem da o id
    ... | ()

Figure 4.5: Proving the totality of the compileCUDA function in Agda. The function’s returning a Bool serves only to illustrate which clauses must and must not be provided.

id : {i : Level} {A : Setᵢ} → A → A
id x
    = x

Combined with the proof da and o, a proof that the Cond operation is used in the tree being examined, this instantiates withRightElem’s type to:

withRightElem : ∀{TS}
    → (da : AllRight CUDACompatible TS)
    → (o : Right Cond ∈ TS)
    → (id : CUDACompatible Cond → CUDACompatible Cond)
    → CUDACompatible Cond

The absurd pattern is valid in the case that there are no constructor patterns possessing the type CUDACompatible Cond, which is obviously the case. The function is thus recognised as total.

Addressing the earlier point regarding the choice to use Agda, GHC’s totality checking is altogether too conservative. In the case of the proof presented here, for example, GHC is unable to verify the safety of the withRightElem function, admitting the erroneous pattern combination:

withRightElem ds (Left ds) Head f
    = ...

failing to recognise that the information gained from both patterns produces the unsolvable constraint Left a ∼ Right b.
4.7 Discussion

In this chapter we have presented what is essentially a design pattern for the type-safe separation of an interface from multiple implementations. Each implementation may enforce different typing requirements and restrictions on a data type without limiting other uses of the same type. While we have presented our solution in Haskell, we have also illustrated an encoding of our methods in Agda, proving some useful safety properties of our technique in the process.

Interestingly, in some situations we are able to support an element of re-use that would ordinarily require the introduction of a higher-rank type, for example when applying multiple functions to the same DSL expression. Even if a higher-rank type is required for another reason, for example to support the same type of re-use in a tagless DSL representation, our method applies equally well. In the next chapter we shall see why this is useful in a practical setting, as we use generic constraints in the development of a DSL for exploiting heterogeneous parallelism in Haskell.
The previous chapter introduced generic constraints for separating the interface of a DSL from the type systems of its individual implementations. In this chapter we shall demonstrate the benefits of generic constraints in a practical setting by building a DSL for heterogeneous parallel programming in Haskell. In particular we shall see how our technique’s main departure from others such as that of Hughes [25]—its compatibility with higher-rank types—allows us to combine multiple implementations of a DSL to create more sophisticated DSL features in a composable fashion. Importantly, while only a subset of the language may be translatable to a given target platform, we shall see that such a subset is described statically and that any associated type-level restrictions do not impact other potential back ends.

5.1 A source language

The DSL to be embedded is a slightly more advanced variant of the TaglessExp class discussed in Chapter 4. Figure 5.1 gives the language’s complete definition. The valueE and condE methods of the class will be used unchanged. The addE and eqE functions, however, will subsumed by a more general notion of unary and binary operators, represented by the associated type families UnOp and BinOp. This layer of abstraction affords us flexibility when deciding what constitutes an operator on a given target platform. A Haskell evaluator, for example, may use a unary or binary function directly, whereas a C code generator will need to select a representation which creates an appropriate AST. In any case, the types UnOp e and BinOp e must at least implement some notion of unary and binary operators respectively, as defined by the TaglessUnOp and TaglessBinOp classes.

The lists of types discussed in Chapter 4 are not so much built as described – the presence of a list as is established before being appropriately constrained. A side-effect of this is that every subexpression in a term will be parameterised by the same list as – only the constraints may differ at a given point. Importantly, this is precisely what is required to overload many of Haskell’s operators to work with values built with our DSL. As an example, consider the Num type class:

```haskell
class Num a where
    (+), (-), (*) :: a -> a -> a
    abs, signum  :: a -> a
    fromInteger :: Integer -> a
```
\begin{align*}
\textbf{class} & \ (\text{TaglessUnOp \ (UnOp} \ e), \ \text{TaglessBinOp \ (BinOp} \ e)) \\
& \Rightarrow \text{TaglessExp \ e \ where} \\
\textbf{type} & \ \text{UnOp} \ e \ :: \ \star \rightarrow \star \rightarrow \star \\
\textbf{type} & \ \text{BinOp} \ e \ :: \ \star \rightarrow \star \rightarrow \star \\
\text{valueE} & \ :: \ (a \in as) \\
& \Rightarrow \ a \rightarrow e \ as \ a \\
\text{unOpE} & \ :: \ (a \in as, \ b \in as) \\
& \Rightarrow \ \text{UnOp} \ e \ as \ (a \rightarrow b) \rightarrow e \ as \ a \rightarrow e \ as \ b \\
\text{binOpE} & \ :: \ (a \in as, \ b \in as, \ c \in as) \\
& \Rightarrow \ \text{BinOp} \ e \ as \ (a \rightarrow b \rightarrow c) \\
& \rightarrow \ e \ as \ a \rightarrow e \ as \ b \rightarrow e \ as \ c \\
\text{condE} & \ :: \ (a \in as, \ \text{Bool} \in as) \\
& \Rightarrow \ e \ as \ \text{Bool} \rightarrow e \ as \ a \rightarrow e \ as \ a \rightarrow e \ as \ a
\end{align*}

\begin{align*}
\textbf{class} & \ \text{TaglessUnOp} \ o \ \text{where} \\
\text{absO} & \ :: \ (\text{Num} \ a, \ a \in as) \Rightarrow o \ as \ (a \rightarrow a) \\
\text{sinO} & \ :: \ (\text{Floating} \ a, \ a \in as) \Rightarrow o \ as \ (a \rightarrow a) \\
\ldots
\end{align*}

\begin{align*}
\textbf{class} & \ \text{TaglessBinOp} \ o \ \text{where} \\
\text{addO} & \ :: \ (\text{Num} \ a, \ a \in as) \Rightarrow o \ as \ (a \rightarrow a \rightarrow a) \\
\text{eqO} & \ :: \ (\text{Eq} \ a, \ a \in as, \ \text{Bool} \in as) \Rightarrow o \ as \ (a \rightarrow a \rightarrow \text{Bool}) \\
\ldots
\end{align*}

Figure 5.1: The TaglessExp source language.
Suppose that we wish to overload the \((+)\) operator on expressions in our DSL, presumably using the `addO` method in our implementation. We seek an instance declaration that allows us to overload the \((+)\) operator, which has type `Num a \Rightarrow a \rightarrow a \rightarrow a`. The required instance now falls out provided we work the necessary `Num a` and `a \in as` constraints into the instance context:

```haskell
instance (TaglessExp e, Num a, a \in as) \Rightarrow Num (e as a) where
 (+) = binOpE addO
```

Extending this procedure appropriately across the Haskell Prelude’s wealth of overloadable functions means that, for the most part, `TaglessExp` values will look just like ordinary Haskell terms. The only evidence of our methodology will be the need to supply an instantiation for each type-level list accumulated and the absence of the methods of the `Eq` or `Ord` classes, for example:

\[
(\equiv) :: Eq a \Rightarrow a \rightarrow a \rightarrow Bool
\]

While \((\equiv)\) is overloaded in the type of its arguments, its result must always be of type `Bool`. Unfortunately, we need the ability to return a value of `e as Bool`, for some `TaglessExp e`. Consequently we shall define our own version of the `Eq` class, `Eq_`:

```haskell
class IsBool b where
 true, false :: b
 not :: b \rightarrow b
 (\&\&), (\lor) :: b \rightarrow b \rightarrow b

class IsBool (BoolFor a) \Rightarrow Eq_ a where
 type BoolFor a :: *
 (\equiv), (\neq) :: a \rightarrow a \rightarrow BoolFor a
```

`IsBool` permits us to overload the standard boolean operators and truth values, while `Eq_` gives us decidable equality. An `Ord_` class (not shown here) is defined similarly, providing methods such as \((<\_\_\_\_\_\_\_\_\_)\) and \((>_\_\_\_\_\_\_\_\_\_\_)\). With these classes, the standard instances of the Haskell Prelude drop out in the obvious way:

```haskell
instance IsBool Bool where
 true_ = True
 false_ = False
 not_ = not
 (\&\&)_ = (\&
 (\lor)_ = (\lor)
```

as do those for tagless expressions, for example:
\[\text{instance} \ (\text{TaglessExp} \ e, \text{Eq} \ a, \ a \in \text{as}, \text{Bool} \in \text{as}) \Rightarrow \text{Eq} \_a \ (e \ as \ a) \ where\]

\[
\begin{align*}
\text{type} \ &\text{BoolFor} \ (e \ as \ a) = e \ as \ \text{Bool} \\
&\text{(≡ₐ)} = \text{binOpE} \ \text{eqO}
\end{align*}
\]

5.2 Haskell in Haskell

The instances of the previous section allow us to take an ordinary-looking Haskell expression or function and interpret it as a term in the TaglessExp language. For example, given a function:

\[
x \mapsto y \mapsto z \mapsto a \mapsto a
\]

we may instantiate \( a \) to the type \((\text{TaglessExp} \ e, \text{Floating} \ a, \ a \in \text{as}) \Rightarrow e \ as \ a\). By subsequently choosing an appropriate TaglessExp instance \( e \) we can target a particular platform. The simplest such platform we can target is Haskell itself, c.f. the evaluate function of Chapter 4:

\[
\text{newtype} \ \text{Identity} \ as \ a \\
= \text{Identity} \ \{ \text{runIdentity} :: a \}
\]

Identity is so-named because a value of type \( \text{Identity} \ as \ a \) is really just a value of type \( a \). Its constructor does not possess a context since Haskell, being the host language for our DSL, is capable of interpreting any value or type we embed within it. This includes function types, meaning that the Identity type can also capture unary and binary operators:

\[
\text{instance} \ \text{TaglessUnOp} \ \text{Identity} \ where \\
\text{absO} = \text{Identity} \ \text{abs} \\
\text{sinO} = \text{Identity} \ \text{sin} \\
\ldots
\]

\[
\text{instance} \ \text{TaglessBinOp} \ \text{Identity} \ where \\
\text{addO} = \text{Identity} \ (+) \\
\text{eqO} = \text{Identity} \ (\equiv) \\
\ldots
\]

\[
\text{instance} \ \text{TaglessExp} \ \text{Identity} \ where \\
\text{type} \ \text{UnOp} \ \text{Identity} = \text{Identity} \\
\text{type} \ \text{BinOp} \ \text{Identity} = \text{Identity} \\
\ldots
\]

The remainder of the type's TaglessExp instance is shown in Figure 5.2. The valueE, unOpE, binOpE and condE functions are all obtained for free due to the fact that Identity \( as \) is the identity (‘do nothing’) applicative functor (McBride and Paterson [41]):\(^1\)

\(^1\)Note that lifting applications in this manner is safe due to the fact that we are embedding a pure subset of the Haskell language.
instance TaglessExp Identity where
  type UnOp Identity = Identity
  type BinOp Identity = Identity
  valueE = pure
  unOpE f x = f ∘ x
  binOpE f x y = f ∘ x ∘ y
  condE = liftA3 (λp t f → if p then t else f)

Figure 5.2: Identity’s TaglessExp instance.

class Functor f ⇒ Applicative f where
  pure :: a → f a
  (⊛) :: f (a → b) → f a → f b
  liftA3 :: Applicative f
          ⇒ (a → b → c → d) → f a → f b → f c → f d
  liftA3 f x y z = pure f ∘ x ∘ y ∘ z

instance Functor (Identity as)
  fmap f (Identity x) = Identity (f x)

instance Applicative (Identity as) where
  pure x = Identity x
  Identity f ⊛ Identity x = Identity (f x)

In order to evaluate an expression, e say, we must impose the correct type on e before invoking runIdentity. For example:

    runIdentity (saxpy 2 3 4 :: Identity '[Float] Float)

which here computes the Haskell value 10.0 :: Float in return.

5.3 Run-time generation of C code

Nikola (Mainland and Morrisett [36]) and Accelerate (Chakravarty et al. [13]) are both examples of DSLs which generate CUDA-compatible C code at run-time. In this section we will develop a TaglessExp instance which is also capable of generating C code. The aim is to be able to write something like the previous section’s saxpy function and have the C code shown in
The saxpy function alongside the C code that might be generated on picking a suitable Floating/TaglessExp instance.

Figure 5.3 generated as a result. Initially we will consider only generating scalar code, as in the transformation shown in Figure 5.3, but we will later build extensions for generating parallel code using OpenMP [59] and streaming SIMD extensions (SSE).

As before, we begin with the type that forms the basis of the TaglessExp instance. The type `C as a` represents a C expression with top-level type `a` which may mention any of the types `as`:

```haskell
newtype C as a = C { runC :: All CTypeable as ⇒ CData }
```

The constraint `All CTypeable as` in C's definition permits a C expression to mention only those types whose values have an equivalent C representation:

```haskell
class Storable a ⇒ CTypeable a where
   typeOfC :: Proxy a → C.Type
   valueC :: a → C.Exp
```

Note that restriction of this kind was not required when defining the `Identity` type of the previous section as Haskell is capable of handling arbitrary types. The types `C.Type` and `C.Exp` are those of the C library used by the quasiquoters of Mainland [35], the members of which will all be distinguished with the prefix C. The superclass constraint `Storable a` is provided by Haskell’s foreign function interface (FFI) and requires that values of type `a` be serialisable to and from the C heap.

For code generation purposes, it will be useful for a value with type `C as a` to contain both its type and equivalent C representation. In the case of the body of the saxpy function defined in Figure 5.3, for example, this will be the pair of ASTs which when pretty-printed yields:

```plaintext
(float, α × x + y)
```

The C type’s field of type `CData` is just such a pairing of type and expression:

```haskell
type CData = (C.Type, C.Exp)
```

Unlike the Identity type of the Haskell evaluator, the C type is not suitable for representing the type of unary and binary operators. This is due to the fact that in C, the addition operator (+) is not a function or expression – it really is an operator. A quotation such as:
makes no sense; in order to apply the operator, its arguments, $e_1$ and $e_2$ say, must be available at the time of application. We can defer the provision of an operator’s arguments by abstracting them as arguments to a function; in the case of the addition operator we have:

$$\lambda e_1. e_2 \rightarrow [C.cexp| e_1 + e_2 ]$$

The CUnOp and CBinOp types generalise this abstraction so that the arguments and result are not expressions but CData pairs; doing so means that the type of an operator’s application may also be computed given a number of input types:

```
newtype CUnOp as a = CUnOp (CData -> CData)
newtype CBinOp as a = CBinOp (CData -> CData -> CData)
```

The instances for TaglessUnOp and TaglessBinOp are as follows:

```
instance TaglessUnOp CUnOp where
  absO = CUnOp (\(\tau, e \rightarrow (\tau, [C.cexp| abs(\exp:e) ]))
  sinO = CUnOp (\(\tau, e \rightarrow (\tau, [C.cexp| sin(\exp:e)]))
...
instance TaglessBinOp CBinOp where
  addO = CBinOp (\(\tau_1, e_1, e_2 \rightarrow (\tau_1, [C.cexp| \exp:e_1 + \exp:e_2 ]))
  eqO = CBinOp (\(\_, e_1, e_2 \rightarrow ([C.cty| \text{int}], [C.cexp| \exp:e_1 == \exp:e_2 ]))
...
```

The unary operators absO, sinO and addO all produce results with the same type as their arguments; in the case of addO we arbitrarily pick the first. The eqO operator always returns a boolean result, which we encode in C using the language’s int type. Expression construction is straightforward thanks to the antiquoter exp, which splices a value of type C.Exp into a C quasiquotation.

Figure 5.4 defines C’s TaglessExp instance. The choice of representation for operators leaves relatively little work for the unOpE and binOpE functions, which simply apply the functions wrapped by the CUnOp and CBinOp constructors to their argument CData. The valueE function is, as usual, verbose but unremarkable, isolating the necessary CTyepable dictionary in order to reflect the type and lifted term of the value being embedded. The condE function is
\begin{verbatim}
instance TaglessExp C where
type UnOp C = CUnOp
type BinOp C = CBinOp
unOpE (CUnOp f) (C d) = C (f d)
binOpE (CBinOp f) (C d1) (C d2) = C (f d1 d2)
condE (C dp) (C dt) (C df) = C $ let (_ , p) = dp
     ( τ , t ) = dt
     ( _, f ) = df
     in ( τ , C.cexp $exp:p ? $exp:t : $exp:f ])
valueE x = valueE' x
where
  valueE' :: (a ∈ as) ⇒ a → C as a
valueE' x = C (withElem (Proxy :: Proxy as) (f x))
  where
    f :: a → Trap CTypeable a → CData
    f x Trap
    = (typeOfC (Proxy :: Proxy a), valueC x)
\end{verbatim}

Figure 5.4: A TaglessExp instance for generating C code for unary, binary and ternary operator applications.
in essence a specialised ternary operator and so uses quasiquoters directly to achieve the same effect as the combinations of CUnOp/unOpE and CBinOp/binOpE. The only caveat is that condE's type:

\[
\text{condE :: } (a \in as, \text{Bool} \in as) \Rightarrow \text{C as Bool} \rightarrow \text{C as a} \rightarrow \text{C as a} \rightarrow \text{C as a}
\]
does not directly afford the context All CTypeable as, which is required if we are to pattern-match on the CData fields contained within its three arguments:

```
newtype C as a = C { runC :: All CTypeable as ⇒ C as Bool → C as a → C as a → C as a }
```

However, such a context is available inside the value we must return. That is to say, in the definition:

\[
\text{condE } (C d_p) (C d_t) (C d_f) = C d_r
\]
the value \( d_r \) has the type All CTypeable as \( ⇒ \) C as a. We are therefore free to pattern-match over the arguments \( d_p, d_t \) and \( d_f \) so long as we do so inside the C constructor; here we do so using `let`-bindings.

### 5.3.1 Compiling Haskell to C

All that is needed now is a function for compiling Haskell functions into their C counterparts. In what follows we omit the details of compiling and dynamically linking code into a running application and focus on code generation, which is arguably the more relevant problem. Our C code generator takes the form of a polyvariadic function (Section 3.5) not unlike that used by Nikola (Mainland and Morrisett [36]):

```
class CCompilable a where
    compileCWith :: [(C.Type, String)] → a → IO C.Func
    compileC :: CCompilable a ⇒ a → IO C.Func
    compileC
        = compileCWith []
```

The `compileCWith` function takes a value of type `a` and a list of the types and names of its free variables, which will correspond to parameters in the resulting C function definition. The result is an IO action producing a C function definition. The IO monad is necessary for dynamic compilation and linking later on. The `compileC` helper function kicks off compilation with an initially empty list of free variables; this acts as an accumulator in the two instances of `CCompilable` (Figure 5.5).

In the base case, `compileCWith`'s argument is a C expression that is ready to be compiled, \( e \), and its type, \( \tau \). A function name is generated using the `newName` function, which accepts a prefix with which to build a globally unique name in the IO monad:
instance \( (a \in as, \text{All C} \text{Typeable as}) \Rightarrow \text{CCompilable} \ (C \ as \ a) \) where

\[
\text{compileCWith} \ f\text{vs} \ (C \ (\tau, \ e)) = \text{newName} \ “f” \Rightarrow \lambda f \rightarrow \\
\text{pure} \ [C.\text{cfun}| \text{extern} \ “C” \ \text{\$ty:}\tau \ \text{\$id:}\ f (\text{\$params:}\ ps) \} \ \\
\text{return} \ \text{\$exp:}\ e; \\
\]

where

\[
ps = \text{map} \ (\lambda (\sigma, \ x) \rightarrow [C.\text{cparam}| \text{\$ty:}\sigma \ \text{\$id:}\ x]) \ f\text{vs}
\]

instance \( (a \in as, \text{All C} \text{Typeable as}, \text{CCompilable} \ r, c \sim C \ as \ a) \Rightarrow \text{CCompilable} \ (c \rightarrow r) \) where

\[
\text{compileCWith} \ f\text{vs} \ f \\
= \text{newName} \ “x” \Rightarrow \lambda x \rightarrow \\
\text{compileCWith} \ ((\tau, \ x) : f\text{vs}) \ (f \ (C \ (\tau, \ [C.\text{cexp}| \text{\$id:}\ x])))
\]

where

\[
\tau = \text{withElem} \ (\text{Proxy :: Proxy as}) \ (\text{trapTypeOfC} \ (\text{Proxy :: Proxy a}))
\]

Figure 5.5: Inductively defining the compileCWith family of functions.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
<th>Argument type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>A C identifier</td>
<td>String</td>
</tr>
<tr>
<td>int</td>
<td>An integer (int) constant</td>
<td>Integral ( a \Rightarrow a )</td>
</tr>
<tr>
<td>ty</td>
<td>A C type</td>
<td>C.Type</td>
</tr>
<tr>
<td>exp</td>
<td>A C expression</td>
<td>C.Exp</td>
</tr>
<tr>
<td>func</td>
<td>A C function definition.</td>
<td>C.Func</td>
</tr>
<tr>
<td>params</td>
<td>A list of C function parameters</td>
<td>[C.Param]</td>
</tr>
</tbody>
</table>

Table 5.1: A subset of the antiquotation specifiers supported by Mainland’s C quasiquoting library, along with their argument types.
newName :: String → IO String

The newly generated name, $f$, is then spliced into the C function definition using the id antiquote specifier (Table 5.1). The values $\tau$ and $e$ form the C function's return type and body and are included via the ty and exp antiquotations. The now-complete accumulated list of free variables, $fvs$ is transformed into a C parameter list (of type [C.Param]) using the cparam quasiquoter, before being passed to the params antiquoter in order to complete the C definition. The pure function here has type:

$$\text{pure} :: a \rightarrow \text{IO } a$$

and is used to lift the pure quotation into the effectful IO monad (recall that compileCWith's result type is IO C.Func).

The inductive instance accepts a function and applies it to a freshly-generated argument, $x$, before recursively compiling the resulting value. The type of the new argument is built by the trapTypeOfC function, which wraps the CTypeable class' typeOfC function to take an explicitly trapped CTypeable dictionary:

$$\text{trapTypeOfC} :: \text{Proxy } a \rightarrow \text{Trap CTypeable } a \rightarrow \text{C.Type}$$

$$\text{trapTypeOfC } p \text{ Trap } = \text{typeOfC } p$$

As discussed in Section 3.5, we have defined the inductive instance using a local functional dependency: the equality constraint $c \sim C \text{ as } a$ in the instance context means that the compiler is free to select the instance even if it is not yet certain that the argument type of the function being compiled unifies with the type $C \text{ as } a$, deferring the unification to the constraint solver.

The code for the C function saxpy shown in Figure 8.3 may now be generated by compiling the Haskell function of the same name:

$$\text{compileC} (\text{saxpy} :: C [\text{Float}] \text{ Float } \rightarrow C [\text{Float}] \text{ Float } \rightarrow C [\text{Float}] \text{ Float} \rightarrow C [\text{Float}] \text{ Float})$$

Once more, we must supply a suitable type for the saxpy function before compileC will accept it as an argument. In return, we receive C code shown in Figure 5.6.

5.3.2 Type-safe parallelisation and optimisation

We will now make use of the scalar code generator compileC in order to generate data-parallel programs. In doing so we will see how higher-order constraints can play a role in providing guarantees about the validity of parallelisation and program optimisation. For simplicity we shall work with streams of data, which we shall represent as plain Haskell lists:

$$\text{newtype Stream } p \text{ as } a$$

$$= \text{Stream } \{ \text{runStream } :: [a] \}$$
extern "C" float f1(float x1, float x2, float x3) {
    return x1 * x2 + x3;
}

Figure 5.6: The C code generated as a result of compiling the saxpy function.

type family Streamable p as :: Constraint

Aside from the element type a, a Stream is parameterised by a target platform p and a set of types as which, as we shall see, must be Streamable on the platform p. In the case of the C back end, for example, the idea is that the Streamable family will be instantiated such that the constraint Streamable C as resolves to the constraint All CT ypeable as. We could take a more direct approach and simply make Stream itself a data family, as in:

data family Stream p :: [⋆] → ⋆

data instance Stream C as a = CStream { runCStream :: All CT ypeable as ⇒ [a] }

Here the CT ypeable constraint has been incorporated specifically into the type of the CStream constructor. This scheme is also more flexible in that it permits platforms to opt out of using lists to represent streams of data. However, such an interface necessarily entails more supporting infrastructure; in the spirit of being brief we shall thus stick with the simpler solution.

The 'hint' that parallelism might be exploited comes in the form of a map-like combinator, which we shall overload so that it can support a variety of targets:

class MappablePlatform p where
    mapE :: (a ∈ as, b ∈ as, Streamable p as) ⇒ (∀e. TaglessExp e ⇒ e as a → e as b) → Stream p as a → Stream p as b

Note that mapE has a rank-2 type: its argument is a function which may be applied to any TaglessExp instance. Figure 5.7 gives the gist of the mapE implementation for the instance MappablePlatform C. The idea is to compile the argument function f using compileC, now modified to return the name of the generated function as well as an abstract syntax tree:

compileC :: CCompilable a ⇒ a → IO (String, C.Func)

A C compilation unit is then built which contains both the code generated by compiling f (fdef) and a wrapper function named g, which applies f to an array xs using a for-loop. The crucial step is the insertion of the OpenMP pragma, which indicates to the C compiler (to be called at run-time) that the loop's iterations are independent of one another and may thus be executed
mapE f s
= ... (fname, fdef) ← compileC f
g ← newName "g"
let u = [C.cunit | $func: fdef
    extern "C" void $id: g($ty: $x + $y, $ty: $z + $y) {
        #pragma openmp parallel for
        for (int i = 0; i < $int: n; ++i) {
            ys[i] = $id: fname(xs[i]);
        }
    }]

...
An SSE vector contains four numbers; knowing that \(xs\) and \(ys\) both have type \(\text{float} \ast\), we can transform this code so that every iteration handles four elements simultaneously:

```c
for (int i = 0; i < 16; i += 4) {
    _mm_store_ps(ys + i,
                 _mm_add_ps(_mm_set1_ps(2), _mm_load_ps(xs + i)));
}
```

Here, the application \(_\text{mm\_load\_ps}(xs + i)\) is used to create a four element vector containing the elements \(xs[i]\), \(xs[i + 1]\), \(xs[i + 2]\) and \(xs[i + 3]\). The call \(_\text{mm\_set1\_ps}(2)\) creates a constant vector where each element is the number 2. These are added together element-wise in parallel by virtue of the intrinsic \(_\text{mm\_add\_ps}\); the resulting vectors elements are then stored in the elements \(ys[i]\), \(ys[i + 1]\), \(ys[i + 2]\) and \(ys[i + 3]\).

This transformation can be realised by an appropriate series of \texttt{TaglessUnOp}, \texttt{TaglessBinOp} and \texttt{TaglessExp} instances which overload, for example, the \texttt{addO} and \texttt{valueE} methods to produce ASTs containing SSE intrinsics, as in:

```haskell
newtype SSEBinOp as a = SSEBinOp (CBinOp as a)
instance TaglessUnOp SSEBinOp where
    addO
    = SSEBinOp \$ CBinOp \$
        \(\lambda (\tau_1, e_1) (\_, e_2) \rightarrow (\tau_1, [\text{C.exp}] \_\text{mm\_add\_ps}([\text{exp}:e_1, \text{exp}:e_2]))\)
```

The key point is that this transformation is only valid if \(xs\) and \(ys\) are arrays of single-precision floating-point elements. But this is precisely the sort of constraint that the \texttt{All} class is able to express:

```haskell
class SinglePrecision a
instance SinglePrecision Float
newtype SSE as a = SSE (All SinglePrecision as \(\Rightarrow\) C as a)
```

We omit the full details of \texttt{TaglessExp/MappablePlatform} instances and associated compilers for the SSE platform here, but suffice it to say that the type-level tools we have developed up to this point are flexible enough to support such back ends.

### 5.3.3 Simultaneous heterogeneous code generation

We can simplify the use of the \texttt{mapE} function by providing our own syntactic sugar:

```haskell
using :: Proxy p \(\rightarrow\) (Stream p' [a] a \(\rightarrow\) b) \rightarrow Stream p' [a] a \(\rightarrow\) b
using
```
data \( (p : ||: q) \) as a

type instance Streamable \( (p : ||: q) \) as
\[ = (\text{Streamable } p \text{ as, Streamable } q \text{ as}) \]

instance \( (\text{MappablePlatform } p, \text{MappablePlatform } q) \)
\[ \Rightarrow \text{MappablePlatform } (p : ||: q) \] where

\[
\text{mapE } f \ s
= \ldots
\]
\[
\text{let } (s_1, s_2) = \text{split } s
\]
\[
t_1 \leftarrow \text{rpar (force (mapE } f \ s_1))
\]
\[
t_2 \leftarrow \text{rpar (force (mapE } f \ s_2))
\]
\[
\ldots
\]
\[
\text{return (merge } t_1 \ t_2)\]

Figure 5.8: Using mapE’s rank-2 type to enable a task-parallel MappablePlatform instance.

\[
= \text{flip const}
\]
\[
c = \text{Proxy }:: \text{Proxy C}
\]
\[
sse = \text{Proxy }:: \text{Proxy SSE}
\]
\[
cuda = \text{Proxy }:: \text{Proxy Accelerate}
\]

The values c, sse and cuda are named proxy arguments, allowing us to write something like, for example:

\[
\text{using sse runStream } (\text{mapE} \ (2+) \ (\text{Stream } [1..10])) :: [\text{Float}]
\]

which adds 2 to each of the elements of a stream containing the floating-point numbers 1 to 10 using the C/OpenMP/SSE back end. Note that the Accelerate type used in the cuda proxy’s definition wraps the Accelerate (Chakravarty et al. [13]) library with TaglessExp and MappablePlatform instances, which we have omitted here for brevity.

As mentioned, mapE exposes an opportunity for data parallelism. In this section we shall see that our DSL can also capture task parallelism, in which a computation is divided into logical units of work which are performed in parallel. This capability results from the ability to combine platform-specific constraints with higher-rank types (see Section 4.4) and enables us to extend our language such that one can write:

\[
\text{using (sse }|||\text{ cuda) runStream } (\text{mapE} \ (2+) \ (\text{Stream } [1..10])) :: [\text{Float}]
\]

which will process half of the elements on the CPU and half on a CUDA-capable GPU. The job of the operator (|||) is simply to create yet another proxy:
Proxy p → Proxy q → Proxy (p ||| q)
Proxy ||| Proxy
  = Proxy

The type p ||| q is the workhorse, defining an instance such as that alluded to in Figure 5.8, whose mapE method invokes the mapE functions of both p and q’s MappablePlatform instances on distinct portions of the input stream. Here we have accomplished this using the Par monad of Marlow et al. [37], partitioning and recombining the input list with the split and merge functions:

\[
\text{split} :: \text{Stream} (p ||| q) as a \rightarrow (\text{Stream} p as a, \text{Stream} q as a)
\]
\[
\text{merge} :: \text{Stream} p as a \rightarrow \text{Stream} q as a \rightarrow \text{Stream} (p ||| q) as a
\]

where split divides a list into two halves and merge concatenates two halves into a new whole. Of course, intelligent implementations of split and merge would likely require more arguments or context. These simplifications serve our purposes, however, and such scheduling considerations lie outside the scope of this thesis. Observe that the two mapE applications illustrate the importance of the argument function f’s higher-rank type – in the composition sse ||| cuda, for example, f’s type will be instantiated twice:

\[
f :: \text{SSE as a} \rightarrow \text{SSE as b}
\]
\[
f :: \text{Accelerate as a} \rightarrow \text{Accelerate as b}
\]

In each case, one half of the constraints paired by (: ||| :)’s Streamable instance is implicitly called upon to perform the necessary mapping.

5.4 Transparent templating and compile-time metaprogramming

Template Haskell’s Exp type (Section 3.7) defines the abstract syntax of Haskell expressions. In this section we develop an instance of the TaglessExp class which targets Exp. Such an instance affords us the ability to stage DSL expressions transparently at compile-time. Importantly, it also illustrates one method for providing a type-safe interface to Template Haskell, which is untyped. Consider:

\[
\text{notTH} :: \text{ExpQ} \rightarrow \text{ExpQ}
\]
\[
\text{notTH} \ x
  = [] \ not (\ x \)
\]
\[
f = $(\text{notTH} [] \text{“False”} \])
\]

Here, the splice in f’s body is well-typed (it has type ExpQ), but the code it produces (the application not “False”) is not:

\[
\text{not} :: \text{Bool} \rightarrow \text{Bool}
\]
\[
\text{“False”} :: \text{String}
\]
There has been much discussion on whether these sorts of problems could and should be avoided by rewriting Template Haskell’s API to enforce a stronger type system, but a cheaper alternative is simply to wrap the existing API with phantom types. The solution developed here is not as comprehensive as the work presented by Thiemann [56], Thiemann and Sulzmann [57] or Kameyama et al. [29], but serves to demonstrate the usefulness of our techniques in an unexpected setting.

We begin once more by defining the type to be made an instance of TaglessExp. In doing so, we shall specify the set of types we wish to capture. The Lift class of Template Haskell represents those types whose values may be lifted into Template Haskell quotations:

```haskell
newtype Template as a = Template { runTemplate :: All Lift as ⇒ ExpQ }
```

The Template type’s TaglessExp instance begins in a similar manner to that of Identity:

```haskell
instance TaglessExp Template where
  type UnOp Template = Template
  type BinOp Template = Template
```

The untyped nature of Template’s underlying ExpQ type means that the type may function as its own unary and binary operators. Implementing the TaglessUnOp and TaglessBinOp classes amounts to placing the appropriate functions in quotations:

```haskell
instance TaglessUnOp Template where
  absO = Template [$ abs $]
  sinO = Template [$ sin $]
  ...

instance TaglessBinOp Template where
  addO = Template [$ (+) $]
  eqO = Template [$ (≡) $]
  ...
```

while the corresponding unOpE, binOpE and condE functions of the TaglessExp instance simply splice these trees into larger quotations describing the necessary applications:

```haskell
unOpE (Template f) (Template x)
  = Template [$ (f) $(x) $]

binOpE (Template f) (Template x) (Template y)
```

\[\text{http://hackage.haskell.org/trac/blog/Template\%20Haskell\%20Proposal}\] (last accessed on 15/07/2012), for example.
class THCompilable a where
    compileTH :: a → ExpQ

instance (a ∈ as, All Lift as, THCompilable r, c ∼ Template as a)
    ⇒ THCompilable (c → r) where
    compileTH f
    = [ λx → $(compileTH (f (Template [[ x ]])))]

Finally, a slight modification to the definitions given for the compileC family of functions produces an analogous Template Haskell compiler, compileTH: Figure 5.9 shows the recursive case responsible for generating Template Haskell expressions representing lambda abstractions. Key is the nested quotation of x after it has been abstracted inside the outer quotation. The contrived example given at the beginning of the section might now be rewritten as:

notTH :: Template as Bool → Template as Bool
    = not *
    f = $(compileTH (notTH “Hello”))

where the literal “Hello” is overloaded, in a similar manner to integers, such that it becomes:

valueE “Hello” :: (String ∈ as) ⇒ Template as String

Since notTH accepts an argument of type Template as Bool, the application will not type check and the type checker will detect an error before any splicing is attempted. In the event that a splice is well-typed, as in $(compileTH (notTH true*)), for example, the underlying ExpQ representations are guaranteed to be well-typed also; these will be extracted by compileTH and spliced into the program.

5.5 The need for type annotations

Somewhat frustrating is the need to supply type annotations whenever a list of types is eliminated, as in Figure 5.10. Here, the fact that runIdentity’s type erases the type as means that we must instantiate as so that the type checker can verify that the constraint (Num a, a ∈ as)
instance (TaglessExp e, Num a, a ∈ as) ⇒ Num (e as a) where

... 2 + 3 :: (TaglessExp e, Num a, a ∈ as) ⇒ e as a
runIdentity :: Identity as a → a

runIdentity (2 + 3 :: Identity \[\text{Int} [\text{Int}]] \text{Int} :: \text{Int}

Figure 5.10: Using type annotations to eliminate lists of types and their associated (∈) constraints.

(as imposed by the Num instance for TaglessExp) is resolvable. In the case of runStream, we avoided this by having the using function ground the types of its arguments:

using :: Proxy p → (Stream p \[\text{a} \text{a} \rightarrow b]) → Stream p \[\text{a} \rightarrow b]

Here, the type of the stream on which using’s argument must operate has been specialised to accept streams whose computation involves a single type, that of the stream elements. For platforms such as C, in which only a finite, enumerable set of types is supported, this approach can be generalised. For example, we can define a function, (↓), which instantiates a given expression’s list of types to encompass every type supported by a given platform:

type family Types p :: [*]

<table>
<thead>
<tr>
<th>type instance</th>
<th>Types C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[\text{Int}, \text{Float}, \text{Double}, \ldots]</td>
</tr>
<tr>
<td>(↓) :: TaglessExp e ⇒ e (Types p) a → Proxy p → e (Types p) a</td>
<td></td>
</tr>
<tr>
<td>(↓)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>= const</td>
</tr>
</tbody>
</table>

Since Types is a type family, it is not injective and the type checker cannot deduce the platform p from its presence alone. The (↓) combinator thus accepts a proxy argument which identifies p. In the case of the expression earlier passed to evaluate, for instance, we can use the proxy c defined in Section 5.3.3 to infer the type:

(2 + 3) ↓ c :: TaglessExp e ⇒ e \[\text{Int}, \text{Float}, \text{Double}, \ldots\] Int

Note that it is (↓)’s type signature that is important here; its implementation (the const function) simply returns its first argument. The proxy argument exists purely to ascertain the platform being targeted and is discarded.

It is also possible to extend (↓) in a polyvariadic fashion, affording the same type-fixing properties to compileC, for example. This technique will not suffice for all targets however. The
Identity type and associated runIdentity function used above, for example, admit any Haskell type and thus possess a list of types which may be instantiated to an infinite number of types. This is particularly annoying when, given a principal type, there is a mechanical translation from its list of ($\in$) constraints to a minimal satisfying list of types. For example, the type:

$$(\text{Int} \in as, \text{Bool} \in as, \text{Float} \in as) \Rightarrow e \ as \ \text{Float}$$

may always be instantiated to:

$$e' \ [(\text{Int}, \text{Bool}, \text{Float}] \ \text{Float}$$

Each constraint of the form $a \in as$ results in the type $a$ being added to the list of types we shall pick. We can encode this type-level function directly, using either GADTs or type families:

```haskell
type family Satisfying (c :: Constraint) :: [*]

type instance Satisfying (a \in as) = '[a]

type instance Satisfying (a \in as, c) = a : Satisfying c
```

Unfortunately, we cannot use such a family due to the way in which constraints ‘float’ to the leftmost position in a type. Consider:

```haskell
satisfy :: TaglessExp e \Rightarrow (c \Rightarrow e \ as \ a) \Rightarrow e (Satisfying c) \ a
```

Supplying the expression $2 + 3$, which has the inferred type $(\text{TaglessExp} \ e, \text{Num} \ a, a \in as) \Rightarrow e \ as \ a$, to satisfy yields the type:

$$satisfy (2 + 3) :: (\text{TaglessExp} \ e, \text{Num} \ a, a \in as) \Rightarrow e (Satisfying c) \ a$$

in which the constraint $c$ is ambiguous. This is precisely the issue identified in Section 4.3, solved there by the Trap data type. However, using the Trap data type required type annotations also, and so this does not solve our problem. Furthermore, even if we could successfully type the satisfy function, we would then have to provide an implementation capable of proving to the compiler that the type $Satisfying \ c$ really does satisfy the constraint $c$.

If we are willing to ask for the compiler’s help, Template Haskell provides a function reify which allows one to obtain information about an identifier at compile-time. This includes an ADT representation of the identifier’s type, from which we can surely implement the translation realised as a type family above. Moreover, since such an instantiation would occur during type checking, the compiler would be able to verify whether or not the types picked satisfy the constraints being eliminated. However, due to some of Template Haskell’s practical limitations, such as not being able to reify names defined in the same module (see the work of Sheard and Peyton Jones [53] for more information), this technique cannot be applied uniformly. A key point also is that reify can only operate on named values. In this respect it would be beneficial to have something similar to C++’s `decltype` operator, which is capable of returning the declared type of an arbitrary expression.
using :: Proxy p → (Stream p ↦ [a] a → b) → Stream p ↦ [a] a → b
sse :: Proxy SSE
cuda :: Proxy Accelerate
runStream :: Stream p as a → [a]
zipWithE :: (MappablePlatform p, a ∈ as, b ∈ as, c ∈ as, Streamable p as) ⇒ (∀ e. TaglessExp e ⇒ e as a → e as b → e as c) → Stream p as a → Stream p as b → Stream p as c
saxpy :: Floating a ⇒ a → a → a

\[ h \, \alpha \, xs \, ys = \text{using (sse ||| cuda) runStream (zipWithE (saxpy \alpha) \, xs \, ys) } \]

Figure 5.11: A function which exploits both heterogeneous parallelism through higher-rank types and specialisation through partial application.

5.6 Discussion

We have developed a DSL capable of targeting a variety of platforms. In particular, we have exploited the ability of our generic constraints to function in the presence of higher-rank types in order to enable users of our DSL to take advantage of both data and task parallelism in a type-safe manner. As Figure 5.11 shows, existing functions with polymorphic types (here saxpy) may be re-used transparently. Moreover, partial application may be used to effect specialisation: in Figure 5.11, the zipWithE function, a two-stream version of mapE much like the Haskell Prelude’s zipWith function is not passed the saxpy function but a partial application where its first argument, the scale factor, has been specialised to h’s argument \( \alpha \). The call \( h \, 3 \, xs \, ys \), for instance, might generate the following code:

```c
extern "C" void g1(float xs, float ys) {
    #pragma openmp parallel for
    for (int i = 0; i < 16; i += 4) {
        _mm_store_ps(y + i,
                     _mm_mul_ps(_mm_set1_ps(3),
                                _mm_add_ps(_mm_load_ps(xs + i), _mm_load_ps(y + i))));
    }
}
```

In this chapter we have for the most part focussed on embedding values, creating a series of first-order languages in which functions were considered only at points of program compilation. In
the next chapter we shall see how we might transparently embed vanilla Haskell functions into DSL terms using higher-order abstract syntax (Section 3.4). In doing so we shall see that by expressing compilation, for example, as just another form of term evaluation we may remove both the duplication exhibited by the CCompilable and THCompilable type classes and the need for type annotations when rank-2 polymorphism is desired. Furthermore, even more complex functions such as those which pattern-match their arguments may be embedded in some cases.
Higher-order embeddings

In Chapters 4 and 5 we demonstrated how we may embed type systems which constrain both the types and operations involved in domain-specific computations. However, much of that presented has been first-order: while we have considered compiling functions (as in Section 5.3.1, for instance) we have not examined how to embed functions into our language. In this chapter we shall see that embedding functions using higher-order abstract syntax (Section 3.7) can allow us to both remove duplication such as that exhibited in the CComposite and THComposite classes of Chapter 5 and in some cases avoid type annotations even in the presence of higher-rank types. Furthermore we shall demonstrate how this can be achieved without losing support for features such as composite types and pattern-matching.

6.1 Higher-order or higher-rank?

In the previous chapter we saw how heterogeneity in the context of tagless encodings necessitates the introduction of higher-rank types. The mapE function of Section 5.3.2, for example, possesses the rank-2 type:

\[
\text{mapE} :: \text{(MappablePlatform } p, a \in as, b \in as, \text{Streamable } p as) \\
\Rightarrow (\forall e. \ \text{TaglessExp } e \Rightarrow e as a \rightarrow e as b) \\
\rightarrow \text{Stream } p as a \rightarrow \text{Stream } p as b
\]

Here, the function to be mapped must be polymorphic with respect to the type of the platform being targeted. This is crucial in the case of the instance MappablePlatform \((p : \| : q)\), which facilitates task parallelism by defining an implementation of mapE which instantiates \(e\) to both \(p\) and \(q\).

As mentioned in Section 2.4, type inference in the presence of higher-rank types is in general undecidable and so functions with such types must be annotated explicitly. This is a moot point in the case of mapE, which is a type class method and so must have its type signature explicitly provided regardless, but can be frustrating in other contexts such as the following:

\[
f e = (\text{runC } e, \text{runTemplate } e)
\]

The function \(f\) attempts to compile an expression \(e\) to both its C and Template Haskell representations. Of course, \(f\) may not be successfully applied unless we first supply an explicit rank-2 type signature, such as:
\[ f :: (\forall e. \text{TaglessExp } e \Rightarrow e \to [\text{Int}] \text{Int}) \to (\text{CData}, \text{ExpQ}) \]

which permits the instantiation of the type \( e \) to both \( C \) and Template. An alternative to providing such type signatures everywhere is to define a type representing expressions which are, by construction, polymorphic, as seen in Section 3.2:

```haskell
newtype Exp as a = Exp { getExp :: \forall e. \text{TaglessExp } e \Rightarrow e as a }
```

\( f \)’s type will now be inferred as:

\[ f :: (\text{All CTympable as } a, \text{All Lift as } a) \Rightarrow \text{Exp as } a \to (\text{CData}, \text{ExpQ}) \]

In many respects \( \text{Exp} \) is a drop-in replacement for \( \text{TaglessExp} \). Figure 6.1 shows how an instance of the \( \text{Num} \) class, for example, can endow overloaded functions and literals with implicitly polymorphic types. In the definition of the function \((+), \) unwrapping the two \( \text{Exp} \) arguments reveals the values \( e_1 \) and \( e_2 \), which have the types:

\[
\begin{align*}
    e_1 &:: \text{(Num } a, a \in a) \Rightarrow (\forall \varepsilon_1. \text{TaglessExp } \varepsilon_1 \Rightarrow \varepsilon_1 as a) \\
    e_2 &:: \text{(Num } a, a \in a) \Rightarrow (\forall \varepsilon_2. \text{TaglessExp } \varepsilon_2 \Rightarrow \varepsilon_2 as a)
\end{align*}
\]

The application \( \text{binOpE addO} \) picks \( \varepsilon_1 \) and \( \varepsilon_2 \) to be the same, equally polymorphic type, before boxing the result with the \( \text{Exp} \) constructor. The \( \text{fromInteger} \) function receives an argument \( x \) of type \( \text{Integer} \). This is converted to a value of type \( a \) using the \( \text{fromInteger} \) function belonging to the instance \( \text{Num } a \) (specified in the instance context) before being lifted with the combination of \( \text{Exp} \) and \( \text{valueE} \).

Making use of \( \text{Exp} \) in typing higher-order functions is not so simple. Suppose we wish to rewrite \( \text{mapE} \)’s type to capture the polymorphic nature of its argument implicitly using the \( \text{Exp} \) type. Naïvely translating the existing type yields a function which accepts an expression transformer:

\[
\begin{align*}
\text{mapE} :: (\text{MappablePlatform } p, a \in a, b \in a, \text{Streamable } p as) &\Rightarrow (\text{Exp as } a \to \text{Exp as } b) \\
&\to \text{Stream } p as a \to \text{Stream } p as b
\end{align*}
\]

Observe that this variant of \( \text{mapE} \) possesses a rank-3 type, as the uses of \( \text{Exp} \) result in two distinct universal quantifications in the type of the functional argument:

\[
(\text{Exp as } a \to \text{Exp as } b) \cong (\forall e_1. \text{TaglessExp } e_1 \Rightarrow (\forall e_2. \text{TaglessExp } e_2 \Rightarrow e_2 as a) \to e_1 as a)
\]

While this is precisely the type of, for example, the application \((2+)\), as given by the \( \text{Num} \) instance in Figure 6.1, such a type cannot be made to work with compilers such as \( \text{compileC} \).
instance (Num a, a ∈ as) ⇒ Num (Exp as a) where

Exp e₁ + Exp e₂
    = Exp (binOpE addO e₁ e₂)

fromInteger x
    = Exp (valueE (fromInteger x))

...
and compileTH. To see why, observe that in order to have compileC accept a function of type \( \text{Exp as } a \rightarrow \text{Exp as } b \), we would have to write a pair of CComilable instances resembling those of Figure 6.2. Unfortunately, this is in fact impossible. While the base case instance is simple enough, needing only to pattern-match off the additional Exp constructor:

\[
\text{compileCWith } f \text{us} (\text{Exp C } (\tau, e)) \]

the recursive instance is not so straightforward. Consider the fragment of the current C as a instance which applies the function being compiled, \( f \), to a new, freshly-named argument before recursively compiling the result:

\[
\text{compileCWith } ((\tau, x) : \text{fus}) (f (\text{C } (\tau, [\text{cexp} | \text{id}:x ])))
\]

\( f \) here has type \( \text{C as } a \rightarrow \text{C as } b \). Since \( \text{as} \) and \( a \) are phantom types in C's definition, creating the required argument of type \( \text{C as } a \) amounts to wrapping a pair consisting of a type and expression with C's constructor. Contrast this with the required Exp instance, in which \( f \) has the type:

\[
f :: \text{Exp as } a \rightarrow \text{Exp as } b
\]

Now, \( f \)'s argument must be a completely polymorphic value; we cannot rely on any properties of a given TaglessExp instance (in this instance C). In other words, the only tools at our disposal are the valueE, unOpE, binOpE and condE functions. None of these is suitable for constructing a freshly-named argument of an arbitrary type: valueE requires a value of type \( \forall \alpha. \alpha \), of which the only inhabitant is \( \bot \), while unOpE, binOpE and condE require existing expressions as their arguments. We are thus unable to complete the instance.

It seems what is instead needed is a notion of higher-order expressions, as in:

\[
\text{mapE} :: (\text{MappablePlatform } p, a \in \text{as}, b \in \text{as}, \text{Streamable } p \text{ as }) \\
\Rightarrow \text{Exp as } (a \rightarrow b) \\
\rightarrow \text{Stream } p \text{ as } a \rightarrow \text{Stream } p \text{ as } b
\]

The mapE function's type now has the correct rank:

\[
\text{Exp as } (a \rightarrow b) \cong (\forall e. \text{TaglessExp } e \Rightarrow e \text{ as } (a \rightarrow b))
\]

However, the argument is now an expression which embeds a function, as opposed to a function over expressions. To see that this is also problematic recall that, currently, the only way to build an argument with such a type is with the valueE function:

\[
\text{valueE} :: (\text{TaglessExp } e, a \in \text{as}) \Rightarrow a \rightarrow e \text{ as } a \\
\text{Exp } (\text{valueE const}) :: ((a \rightarrow b \rightarrow a) \in \text{as}) \Rightarrow \text{Exp as } (a \rightarrow b \rightarrow a)
\]
In the case of the C back end, the latter expression's type will eventually demand an instance CTypeable \((a \to b \to a)\), which doesn't exist. Despite this, the const function is compilable to C, at least under the previous scheme:

\[
\text{instance } (a \in \text{as}, \text{All CTypeable as}) \Rightarrow \text{CCompilable (C as a) where}
\]

\[
\ldots
\]

\[
\text{instance } (a \in \text{as}, \text{All CTypeable as, CCompilable r, c }\sim\text{ C as a})
\Rightarrow \text{CCompilable (c }\to\text{ r) where}
\]

\[
\ldots
\]

As defined in the previous chapter (Figure 5.5), these CCompilable instances require that only the individual function argument and result types be CTypeable; the function type as a whole is not so constrained because all C function calls are saturated and it is thus never the case that a function type may result. In order to solve this problem, we must provide another way to embed functions into our DSL and make them a first-class component of our language.

### 6.2 First-class embedded functions

As demonstrated in Section 3.4, an elegant technique for embedding Haskell functions into a DSL is higher-order abstract syntax (HOAS). Using HOAS we shall extend the TaglessExp class as follows:

\[
\text{class } (\text{TaglessUnOp (UnOp } e), \text{TaglessBinOp (BinOp } e))
\Rightarrow \text{TaglessExp } e \text{ where}
\]

\[
\ldots
\]

\[
\text{lamE :: (e as a }\to\text{ e as b) }\to\text{ e as (a }\to\text{ b)}
\]

We note that we are once more using the exotic term-permitting 'pseudo-HOAS' discussed in Section 3.4. The lamE method lifts a function over expressions into an expression with a function type. Using it we can now embed the const function in an alternative fashion:

\[
\text{Exp (lamE (}\lambda x \to \text{lamE (}\lambda y \to \text{const } x y)))) \::\text{ Exp as (a }\to\text{ b }\to\text{ a)}
\]

producing an expression which can be passed to the mapE function. Note that classes such as CCompilable and THCompilable can no longer be used to recursively pick off and compile functions of arbitrary arity, for such functions are now embedded in Exp expressions:

\[
\text{compileC :: (C as a }\to\text{ C as b }\to\text{ C as a) }\to\text{ IO C.Func}
\]

\[
\text{const :: a }\to\text{ b }\to\text{ a}
\]

\[
\text{Exp (lamE (}\lambda x \to \text{lamE (}\lambda y \to \text{const } x y)))) \::\text{ Exp as (a }\to\text{ b }\to\text{ a)}
\]

This is now the job of the implementation of lamE, which may observe the abstraction of an argument. That is to say, it is the body of an implementation of lamE has knowledge of the argument being introduced.

107
Unlike the CCompilable and THCompilable instances being replaced, however, lamE’s type does not constrain either of the types \(a\) or \(b\). Two options for fixing this are as follows:

1. **Require that \(a \in as\) and \(b \in as\)**

   Perhaps the most obvious approach is to blindly constrain \(a\) and \(b\), as in:

   \[
   \text{lamE} :: \text{(TaglessExp } e, a \in as, b \in as) \\
   \Rightarrow (e as a \rightarrow e as b) \rightarrow e as (a \rightarrow b)
   \]

   This works well when embedding unary functions:

   \[
   \text{lamE id} :: \text{(TaglessExp } e, a \in as) \Rightarrow e as (a \rightarrow a)
   \]

   but takes no account of Haskell’s currying of higher-arity functions:

   \[
   \text{lamE } (\lambda x \rightarrow \text{lamE} \ (\lambda y \rightarrow \text{const} \ x \ y)) \\
   :: \text{(TaglessExp } e, a \in as, (b \rightarrow a) \in as, b \in as) \\
   \Rightarrow e as (a \rightarrow b \rightarrow a)
   \]

   Here, it is required that the function type \(b \rightarrow a\) be in the list of types \(as\). There is nothing inherently ‘wrong’ with this, but it is often the case that a language designer only wishes to work with saturated (i.e. not partially applied) function applications. This is the case for both the CCompilable and THCompilable classes where, for example, it is the constraints CTypeable \(a\) and CTypeable \(b\) and not CTypeable \((a \rightarrow b)\) which govern whether or not a function is compilable. A more desirable type would thus be:

   \[
   \text{lamE } (\lambda x \rightarrow \text{lamE} \ (\lambda y \rightarrow \text{const} \ x \ y)) \\
   :: \text{(TaglessExp } e, a \in as, b \in as) \Rightarrow e as (a \rightarrow b \rightarrow a)
   \]

2. **Require only that \(a \in as\)**

   We can avoid falling foul of currying by not constraining the result type of the function being embedded at all:

   \[
   \text{lamE} :: \text{(TaglessExp } e, a \in as) \Rightarrow (e as a \rightarrow e as b) \rightarrow e as (a \rightarrow b)
   \]

   This produces arguably more desirable types:

   \[
   \text{lamE } (\lambda x \rightarrow \text{lamE} \ (\lambda y \rightarrow \text{const} \ x \ y)) \\
   :: \text{(TaglessExp } e, a \in as, b \in as) \Rightarrow e as (a \rightarrow b \rightarrow a)
   \]

   \[
   \text{lamE } (\lambda x \rightarrow \text{lamE} \ (\lambda y \rightarrow x \equiv a)) \\
   :: \text{(TaglessExp } e, Eq \ a, a \in as, \text{Bool } \in as) \Rightarrow e as (a \rightarrow a \rightarrow \text{Bool})
   \]

   However this is arguably down to luck. In the first case, the result type \(a\) also appears as an argument type, at which point the constraint \(a \in as\) is picked up. In the second, it is the
Figure 6.3: Rewriting CCompilable’s recursive instance as an implementation of lamE. The base case forms part of the (now non-overloaded) compileC function.

The second approach proves suitable for rewriting both the CCompilable and THCompilable classes of Sections 5.3.1 and 5.4. Figure 6.3 shows how the CCompilable class may be rewritten using the C type’s implementation of lamE. We see that the CData type must be modified to track the free variables once accumulated by the helper function compileCWith. Moreover, the need to generate fresh names for such variables inside lamE requires modifying C to wrap its CData field with a monadic type constructor capable of such generation. Here we have used IO only for the sake of matching the original implementation of compileC; a simple state
monad would have sufficed. The argumentType function reflects the argument type of a Haskell function into a C type:

```haskell
argumentType :: (a ∈ as, All CT ypeable as) ⇒ (C as a → C as b) → C.Type
```

argumentType f

```haskell
= withElem (Proxy :: Proxy as) g
  where
    g :: Trap Typeable a → String
    g Trap
      = show (typeOfC (Proxy :: Proxy a))
```

The C implementation of lamE does not rely on the context $b ∈ as$ as all the information it requires about $b$ is returned from the result of applying the argument function $f$. Moreover it cannot adhere to the constraint $b ∈ as$ unconditionally as CT ypeable does not support function types. In the event that we wish to target a platform such as C and a platform that requires the constraint $b ∈ as$ then, a third, more complex approach is required which combines the merits of the two discussed above without suffering the same drawbacks.

### 6.3 Optionally constraining function types

What we want is a means of constraining the type $b$ in a type $a → b$ only if it is not itself a function type. In order to do so we need a method for recognising whether or not $b$ is an arrow type. An attempt to encode this type-level predicate as a type family doesn't work:

```haskell
type family IsFunction (a :: ⋆) :: Bool

type instance IsFunction (c → d) = True

type instance IsFunction a = False
```

In Haskell, type family instance selection does not proceed in a top-down manner as with value-level pattern-matching and the two instances therefore overlap since the type $a$ unifies with the type $c → d$. This is not permitted and we must therefore base our solution on type classes where overlapping instances may be declared provided that there is always a most-specific definition (see Section 2.1 for more information):

```haskell
class IsFunction (p :: Bool) (a :: ⋆)

instance (p ~ True) ⇒ IsFunction p (c → d)
instance (p ~ False) ⇒ IsFunction p a
```

The use of local functional dependencies in the instance contexts mirrors that of Section 3.5.2. The idea is that the compiler will select an instance regardless of whether it can instantiate $p$ to
True or False at a particular point, deferring necessary unification to the constraint solver after the fact.

Given a context IsFunction \( p \ a \), \( p \) will unify with the type True if and only if \( a \) is a function type. The idea now is to use \( p \) to decide whether or not a constraint of the form \( b \in as \) should be introduced for some type \( b \) and type-level list \( as \). As an example, Section 3.6.2 introduced Potential, a DSL for writing 64-bit x86 assembly programs which made use of optional class constraints:

```haskell
data ClassConstraintsOn
data ClassConstraintsOff
class MaybeHasSZ d s f
instance HasSZ d s ⇒ MaybeHasSZ d s ClassConstraintsOn
instance MaybeHasSZ d s ClassConstraintsOff
```

Here, the MaybeHasSZ class demands a HasSZ context only when the flag \( f \) is instantiated to the type ClassConstraintsOn. GHC’s support for constraint kinds (Section 2.7) actually permits the abstraction of this pattern:

```haskell
class p ? c
instance c ⇒ True ? c
instance False ? c
```

As stated, we are only interested in uses of the form \( p ? (b \in as) \); we will however use the (?) class, showing later how it may be enhanced for general use in type-restricted programming. For now, let us complete the type of \( \text{lamE} \):

```haskell
\text{lamE} :: (\text{TaglessExp} e, a \in as, \text{IsNotFunction} p b, p ? (b \in as))
∂ Proxy p
⇒ (e as a → e as b)
⇒ e as (a → b)
```

Since the point is to introduce the constraint \( b \in as \) if and only if \( b \) is not a function type, we have used a class IsNotFunction which inverts the instances of IsFunction in the obvious manner. The additional argument of type Proxy \( p \) witnesses the instantiation of the type \( p \), which would otherwise be ambiguous:

```haskell
\text{lamE} (\text{Proxy} :: \text{Proxy True})
:: (\text{TaglessExp} e, a \in as, b \in as)
⇒ (e as a → e as b) → e as (a → b)
\text{lamE} (\text{Proxy} :: \text{Proxy False})
:: (\text{TaglessExp} e, a \in as)
⇒ (e as a → e as b) → e as (a → b)
```
6.3.1 Pretty-printing functions

Making use of the knowledge now provided by lamE’s type requires some more work. As a
trivial example, we will build a pretty-printer which annotates the argument and result types
of a function. For example we would like the function \( \lambda (a :: \text{Int}) (b :: \text{Int}) \to a + b \) to be
pretty-printed as:

\[
(\lambda (a :: \text{Int}) \to (\lambda (b :: \text{Int}) \to (a + b :: \text{Int}))
\]

To do so we shall make use of Haskell’s Typeable class, which allows us to reflect a type at
run-time:

```haskell
class Typeable a where
  typeOf :: a \to TypeRep
```

where TypeRep is some concrete representation of a ground type possessing a sensible Show in-
stance. For instance, `show (typeOf (⊥ :: Int))` returns the string “Int” as we might hope. As this
eexample shows, `typeOf` does not examine its argument, which is akin to the type-fixing proxies
we have been using throughout this thesis. The type to be made an instance of TaglessExp is
Pretty, which extends both the constraints Show and Typeable over its phantom list of types
as:

```haskell
newtype Pretty as a = Pretty { runPretty :: (All Show as, All Typeable as) \to [String] \to String }
```

Pretty wraps a function which accepts a list of variable names that may be used and returns a
pretty-printed expression. For example, the outputs:

\[
(\lambda (a :: \text{Int}) \to (\lambda (b :: \text{Int}) \to (a + b :: \text{Int}))
\]

\[
(\lambda (x :: \text{Int}) \to (\lambda (y :: \text{Int}) \to (x + y :: \text{Int}))
\]

might be generated by pretty-printing a term \( p \) with the lists [“a”, “b”] and [“x”, “y”] respec-
tively. Pretty’s lamE implementation is given below:

```haskell
instance TaglessExp Pretty where
  \ldots
  lamE pr f
  = Pretty $ \lambda (v :: vs) \to
    let p = f (Pretty (const v))
    in  "(\lambda (" ++ v ++ " :: " ++ argumentType f ++ ")" ++
       " \to " ++ runPretty p vs ++ resultType f ++ ")"
```
Recall that the new argument to lamE, pr, is a proxy that fixes whether or not the function f returns another functional term. The argumentType function is that used in the C implementation, except that it uses the typeOf function from the Typeable class as opposed to CTypeable's typeOfC:

\[
\text{argumentType} :: (a \in as, \text{All Typeable as}) \\
\Rightarrow (\text{Pretty as } a \rightarrow \text{Pretty as } b) \rightarrow \text{String}
\]

\[
\text{argumentType } f \\
= \text{withElem (Proxy :: Proxy as)} \ g
\]

where

\[
g :: \text{Trap Typeable } a \rightarrow \text{String}
\]

\[
g \text{ Trap} \\
= \text{show (typeOf (⊥ :: a))}
\]

The resultType function names the type of the co-domain of its argument. Given the above definition of argumentType, we would expect to give resultType the type:

\[
\text{resultType} :: (b \in as, \text{All Typeable as}) \\
\Rightarrow (\text{Pretty as } a \rightarrow \text{Pretty as } b) \rightarrow \text{String}
\]

with a correspondingly similar implementation. However, we don't know that the constraint \( b \in as \) will be available – it all depends on whether or not \( b \) is a function type.

### 6.3.2 Constraint-dependent computation

We require a wrapper function that accepts a function like resultType and applies it only in the event that an optional constraint is available. Such a function must reside in the (?) type class:

\[
\text{class } p \ ? \ c \ \text{where} \\
\text{given :: Proxy } p \rightarrow (\text{Dict } c \rightarrow a) \rightarrow \text{Maybe } a
\]

The function given accepts both a proxy value which ascertains whether or not the context \( c \) is available and a function which may only be applied in the presence of such a context. As in Section 4.3, the constraint must be reified as a dictionary (there Trap), though this time there is no need for the separation of constraint constructor and argument:

\[
\text{data Dict } c \ \text{where} \\
\text{Dict :: } c \Rightarrow \text{Dict } c
\]

Indeed, we might rename Dict to Dict₀ and Trap to Dict₁. The two implementations of given fall out by virtue of the instance contexts:

\[
\text{instance } c \Rightarrow \text{True } ? \ c \ \text{where} \\
\text{given } f
\]
= Just (f Dict)

instance False ? c where
  given _ _
  = Nothing

In the case where \( p \) is True, we require a superclass context stating that the constraint \( c \) is actually available. The answer is then Just the result of applying \( f \) to a dictionary trapping the constraint \( c \) (of type Dict \( c \)). If \( p \) is False, the context \( c \) is not resolvable and so we return Nothing. We can use the given function to rewrite Pretty's lamE implementation as follows:

\[
\text{instance TaglessExp Pretty where}
\]
\[
\ldots
\]
\[
lamE \ pr \ f
\]
\[
= \text{Pretty } \lambda (v : vs)
\]
\[
\text{let } p = f (\text{Pretty} (\text{const} \ v))
\]
\[
r = \text{case } \text{given } pr \ (\text{resultType } f) \text{ of}
\]
\[
\text{Just } t \rightarrow " :: " + t
\]
\[
\text{Nothing } \rightarrow ""
\]
\[
in "(\lambda(" + v + " :: " + \text{argumentType } f + ") +
\]
\[
" + \text{runPretty } p \ vs + r + ")"
\]

The proxy value ascertaining whether or not the constraint is available is precisely that given as an argument to \( \text{lamE} \), namely \( pr \). The resultType function now accepts an explicit dictionary encoding the fact that \( b \) is an element of \( as \):

\[
\text{resultType } :: \text{All Typeable as}
\]
\[
\Rightarrow (\text{Pretty } as \ a \rightarrow \text{Pretty } as \ b) \rightarrow \text{Dict } (b \in as) \rightarrow \text{String}
\]
\[
\text{resultType } f \text{ Dict}
\]
\[
= \text{withElem } (\text{Proxy } :: \text{Proxy as}) \ g
\]
\[
\text{where}
\]
\[
g :: \text{Trap Typeable } b \rightarrow \text{String}
\]
\[
g \text{ Trap}
\]
\[
= \text{show } (\text{typeOf } (\bot :: b))
\]

It is worth noting that the use of given and resultType could be avoided altogether by extending the definition of Pretty slightly:

\[
\text{newtype Pretty } as \ a
\]
\[
= \text{Pretty } \{ \text{runPretty } :: (\text{All Show } as, \text{All Typeable } as)
\]
\[
\Rightarrow [\text{String}] \rightarrow (\text{String, String})
\]
\[
\}
\]
Here, the enclosed function returns a pair consisting of a pretty-printed string and the type of the pretty-printed expression. Now the desired result type will be provided as part of applying the argument function \( f \), as in:

\[
\text{instance} \ \text{TaglessExp} \ \text{Pretty} \ \text{where}
\]

\[
\begin{align*}
\text{lamE} \ p \ f \\
&= \text{Pretty} \ \$ \ \lambda (v : vs) \rightarrow \\
&\quad \text{let} \ (e, \tau) = \text{runPretty} \ (f \ (\text{Pretty} \ (\text{const} \ v))) \\
&\quad \ldots \\
&\quad \ldots
\end{align*}
\]

However, the body of \( \text{lamE} \) now has no idea whether or not the type represented by the string \( \tau \) is that of the final result and must print it regardless. This leads to ‘over-printing’, as in:

\[
(\lambda (a :: \text{Int}) \rightarrow (\lambda (b :: \text{Int}) \rightarrow (a + b) :: \text{Int}) :: (\text{Int} \rightarrow \text{Int}))
\]

### 6.4 Polyvariadic functions and quasiquotation

The HOAS approach to embedding functions is syntactically more invasive, as we have seen above. As an example, contrast:

\[
\text{compileC} \ (+)
\]

with:

\[
\text{compileC} \ (\text{Exp} \ (\text{lamE} \ (\lambda x \rightarrow \text{lamE} \ (\lambda y \rightarrow x + y))))
\]

The \( \text{lamE} \) function can only abstract one variable at a time, delegating further abstraction to nested uses of the same function. Furthermore, we must manually box our terms with the \( \text{Exp} \) constructor. These encumbrances are however minor compared to that caused by taking into account the constrained type assigned to \( \text{lamE} \) in the previous section:

\[
\text{lamE} :: (a \in as, \text{IsNotFunction} \ p \ b, p \ ? \ (b \in as))
\]

\[
\Rightarrow \text{Proxy} \ p
\]

\[
\Rightarrow (e \ as \ a \rightarrow e \ as \ b)
\]

\[
\Rightarrow e \ as \ (a \rightarrow b)
\]

which causes our example to grow into the rather unpleasant:

\[
\text{Exp} \ (\text{lamE} \ (\text{Proxy} :: \text{Proxy} \ False) \\
(\lambda x \rightarrow \text{lamE} \ (\text{Proxy} :: \text{Proxy} \ True) \ (\lambda y \rightarrow x + y)))
\]
Here the outer application of \( \text{lamE} \) returns an expression representing a function. Its proxy argument is thus of type \( \text{Proxy False} \), signifying that a constraint of the form \( b \in \text{as} \) should not be introduced as \( b \) is a function type. Conversely, the inner application of \( \text{lamE} \) is supplied with a proxy of type \( \text{Proxy True} \), indicating that its result is not a function and that its type should be constrained.

While the introduction of such proxy arguments is tedious, we are fortunate that, as in Section 3.5, here too we can build a polyvariadic version of \( \text{lamE} \) which removes the need for such nesting. We will use the same \( \text{Variadic} \) class, which we reproduce here:

```haskell
class Variadic a r s | s -> a where
    polyLamE :: (a -> r) -> s
```

The base case instance directly invokes \( \text{lamE} \) as we might expect:

```haskell
instance (TaglessExp e, a ∈ as, b ∈ as, f ∼ (a → b)) ⇒ Variadic (e as a) (e as b) (e as f) where
    polyLamE = lamE (Proxy :: Proxy True)
```

Moreover, the instance removes the need for users to supply proxy arguments by providing an appropriate value as part of \( \text{polyLamE} \)'s implementation. Here an argument of type \( \text{Proxy True} \) enforces that \( b \) is not a function type by introducing the context \( \text{IsNotFunction True} \ b \). The recursive instance meanwhile demands that \( b \) is a function type by picking an argument of type \( \text{Proxy False} \):

```haskell
instance (TaglessExp e, a ∈ as, Variadic b r s, s ∼ e as (c → d), f ∼ (a → c → d)) ⇒ Variadic (e as a) (b → r) (e as f) where
    polyLamE f = lamE (Proxy :: Proxy False) (λx → polyLamE (f x))
```

Here the superclass context requires only that the argument type \( a \) be an element of the list \( \text{as} \): we know that the base case will eventually constrain the non-functional result type. The two local functional dependencies (Section 3.5.2) again permit the type checker to select the instance with a minimum of knowledge about the type variable \( f \). In fact, the \( \text{Exp} \) constructor, which was seen as a burden only moments ago, is now all that is needed in order to have a correct type inferred:

```haskell
polyLamE (+) :: (Num a, Variadic a (a → a) s) ⇒ s
Exp (polyLamE (+)) :: (Num a, a ∈ as) ⇒ Exp as (a → a → a)
```

The application of \( \text{Exp} \) requires that the type variable \( s \) be of the form \( e \ as \ a \), for some \( \text{TaglessExp} \ e \). Thanks to our tactical use of local functional dependencies, GHC’s constraint
solver can then completely infer the desired type. This is arguably a better return than we could have hoped for. In making it possible to target the embedded (+) function at multiple platforms simultaneously, we have also removed much of the hassle in typing its embedding. You might wonder if we can do one better, and provide a function anyPolyLamE which acts as the composition of Exp and polyLamE such that:

\[
\text{anyPolyLamE} (+) :: (\text{Num} \ a, a \in \text{as}) \Rightarrow \text{Exp} as (a \rightarrow a \rightarrow a)
\]

Unfortunately, while the implementation is straightforward:

\[
\text{anyPolyLamE} = \text{Exp} \cdot \text{polyLamE}
\]

anyPolyLamE’s type is not something that GHC will understand. Intuitively, the composition of the two functions may be typed:

\[
\text{Exp} \cdot \text{polyLamE} :: \text{Variadic} b r (\forall e. \text{TaglessExp} e \Rightarrow e \text{ as} a) \\
\Rightarrow (b \rightarrow r) \rightarrow \text{Exp as} a
\]

The ‘higher-rank’ constraint \(\text{Variadic} b r (\forall e. \text{TaglessExp} e \Rightarrow e \text{ as} a)\) cannot be resolved. We might edge closer to a solution by reifying the constraint as a dictionary ourselves, viz.:

\[
\text{anyPolyLamE} :: (\forall e. \text{TaglessExp} e \Rightarrow \text{Dict} (\text{Variadic} b r (e \text{ as} a))) \\
\rightarrow (b \rightarrow r) \rightarrow \text{Exp as} a
\]

which will at least type check. However, we are then unable to produce a Dict of the correct type. In any case, an anyPolyLamE function that requires an additional argument is arguably no cleaner than the explicit composition of Exp and polyLamE. A rather cheekier solution that we can implement is to use Template Haskell to generate the composition of Exp and polyLamE wherever it is needed. Specifically, we shall define a quasiquoter \([\lambda|\cdot|]\) such that:

\[
[\lambda|f|] \equiv \text{Exp} (\text{polyLamE} f)
\]

As mentioned in Section 3.7, quasiquoters may be used in expressions, types, declarations and patterns. The definition of a quasiquoter consists of providing a record of four functions to this effect:

\[
\textbf{data QuasiQuoter} = \text{QuasiQuoter} \{ \text{quoteExp} :: \text{String} \rightarrow \text{ExpQ}, \text{quotePat} :: \text{String} \rightarrow \text{PatQ}, \text{quoteType} :: \text{String} \rightarrow \text{TypeQ}, \text{quoteDec} :: \text{String} \rightarrow \text{DecsQ} \}
\]

The quasiquoter \([\lambda|\cdot|]\) was used above to produce an expression. We shall also define a type quasiquoter which allows us to write types more easily as follows:
blackScholes :: (Floating a, Ord a, Conditional a)
⇒ a → a → a → a → a → a

blackScholes s t r v = s × normCDF d₁ - x × exp (−r × t) × normCDF d₂

where
  d₁ = (log (s / x) + (r + v × v / 2) × t) / (v × sqrt t)
  d₂ = d₁ - v × sqrt t

normCDF :: (Floating a, Ord a, Conditional a) ⇒ a → a
normCDF x = x <₧ 0 ? (1 - w, w)

where
  w = 1 - 1 / sqrt (2 × π) × exp (−l × l / 2) × p k
  k = 1 / (1 + 0.2316419 × l)
  l = abs x
  p = horner cs
  cs = [0.0, 0.31938153, -0.356563782, 1.781477937, -1.821255978, 1.330274429]

horner :: Num a ⇒ [a] → a → a
horner cs z = foldr1 (λ x y → y × z + x) cs

Figure 6.4: A fully overloaded implementation of the Black-Scholes formula for option pricing which may be lifted to operate over values of type Exp using polyLamE.

<table>
<thead>
<tr>
<th>a is not a function type</th>
<th>$\lambda \ a \ a \ \equiv \ Exp \ a \ a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda \ a \ \equiv \ Exp \ a$</td>
<td>$[\lambda \ b \ \equiv \ Exp \ as \ r$</td>
</tr>
<tr>
<td>$\lambda \ a \ → \ b \ \equiv \ Exp \ a \ : \ as \ (a \ → \ r)$</td>
<td></td>
</tr>
</tbody>
</table>

Now even the most substantial terms may be lifted and typed with ease. Figure 6.4 defines a fully overloaded implementation of the Black-Scholes formula for option pricing. Using the expression and type quasiquoters defined above, it can be typed (using Floats as the argument and result types, for example) as follows:

$[\lambda \ blackScholes \ ] :: [\lambda \ Float \ → \ Float \ → \ Float \ → \ Float \ → \ Float \ → \ Float]$
6.5 Composite data types

Up until this point we have only considered embedding terms which manipulate primitive types such as Int or Bool. As we have demonstrated throughout this thesis however, Haskell’s type system is far richer. Parameterised ADTs such as the Maybe, Either, (,) and [] type constructors, for example, are common to many Haskell programs, yet are not currently supported by the techniques we outlined. In this section we shall look at some of the challenges encountered when typing functions which make use of these richer ‘composite’ types, and see what adaptations can be made to overcome them.

6.5.1 Recovering pattern-matching with type families

Pattern-matching is key to writing succinct Haskell programs. Unfortunately, the quasiquoter [\(\lambda\) | ·] and the type-level machinery that empowers it don’t support lifting functions which use pattern-matching. Take the swap function, for instance:

\[
\text{swap} :: (a, b) \rightarrow (b, a) \\
\text{swap} (x, y) = (y, x)
\]

The quotation \([\lambda\text{ swap }]\) cannot be typed because it desugars to the term:

\[
\text{Exp (lamE (\lambda (x, y) \rightarrow (y, x)))}
\]

and there is no way for compiler to unify the types \((a, b)\) and \((a \text{ as } c, b \text{ as } c)\), as required by both the argument and result type of \(\text{lamE}\). Changing \(\text{lamE}\)’s type to accept \((a \text{ as } c, b \text{ as } c)\) is clearly not an option lest we restrict \([\lambda | \cdot]\) to only lift pair-processing functions. One solution, as used by, for example, Accelerate (Chakravarty et al. [13]), is to extend the TaglessExp language to support tuples explicitly:

\[
\text{class (TaglessUnOp (UnOp e), TaglessBinOp (BinOp e))} \\
\quad \Rightarrow \text{TaglessExp } e \text{ where}
\]
\[
\ldots
\]
\[
\text{pairE :: (a \in as, b \in as)} \\
\quad \Rightarrow e \text{ as } a \rightarrow e \text{ as } b \rightarrow e \text{ as } (a, b)
\]
\[
\text{fstE :: (a \in as, b \in as)} \\
\quad \Rightarrow e \text{ as } (a, b) \rightarrow e \text{ as } a
\]
\[
\text{sndE :: (a \in as, b \in as)} \\
\quad \Rightarrow e \text{ as } (a, b) \rightarrow e \text{ as } b
\]

This doesn’t really solve the problem, however. We simply avoid the issue by rewriting functions such as swap to replace pattern-matching with applications of the \(\text{fstE}\) and \(\text{sndE}\) projections:
swapE :: TaglessExp e ⇒ e as (a, b) → e as (b, a)

swapE p
  = pairE (sndE p) (fstE p)

Moreover, this defeats the purpose of being able to use existing data types and functions such as (,) and swap without modifying either the types involved or our language. In essence, this is the same issue as was discussed in Section 3.6.1 when examining Kansas Lava (Gill et al. [23, 24]). – we must pick between a pair of incompatible types and implementations:

swap :: TaglessExp e ⇒ (e as a, e as b) → (e as b, e as a)

where the first cannot be lifted by lamE and the second inhibits pattern-matching. In order to solve this problem the Unpacked type family was introduced, which converts between the two representations (Gill et al. [24]). It turns out that we can play a similar trick. Consider the type family Rep, which maps TaglessExp-level expression types into representations more amenable to pattern-matching:

type family Rep e as a :: *

- type instance Rep e as Bool = e as Bool
- type instance Rep e as Int = e as Int
  ...
- type instance Rep e as [a] = [Rep e as a]
- type instance Rep e as (a, b) = (Rep e as a, Rep e as b)
  ...

We can now rewrite lamE to accept swap as an argument thus:

lamE :: (a ∈ as, IsNotFunction p b, p ? (b ∈ as))
  ⇒ Proxy p
  → (Rep e as a → Rep e as b)
  → e as (a → b)

Figure 6.5 illustrates how lamE and swap’s types can be successively specialised until they are unifiable. The key point is that the return type of lamE is not of something of the form Rep e as (a → b). If it were, the type checker would have no knowledge of e, as, a and b (type families are not injective) and we would have to provide proxy arguments (or similar) to this end. As it stands, lamE builds an expression. Its type fixes the types e, as, a and b and the compiler is subsequently able to type check the application.

Note that the cost of this solution is type inference: the introduction of type families means that any types inferred by the compiler will contain equality constraints that it is unable to solve without an instantiating type annotation:
\[
 \text{lamE} :: (\text{TaglessExp } e, a \in \text{as}, b \in \text{as}) \\
 \Rightarrow \text{Proxy True} \\
 \Rightarrow (\text{Rep } e \text{ as } (a, b) \rightarrow \text{Rep } e \text{ as } (b, a)) \\
 \Rightarrow e \text{ as } ((a, b) \rightarrow (b, a)) \\
\]

\[
\text{lamE} :: (\text{TaglessExp } e, a \in \text{as}, b \in \text{as}) \\
\Rightarrow \text{Proxy True} \\
\Rightarrow ((\text{Rep } e \text{ as } a, \text{Rep } e \text{ as } b) \rightarrow (\text{Rep } e \text{ as } b, \text{Rep } e \text{ as } a)) \\
\Rightarrow e \text{ as } ((a, b) \rightarrow (b, a)) \\
\]

\[
\text{swap} :: (a, b) \rightarrow (b, a) \\
\text{swap} :: \text{TaglessExp } e \\
\Rightarrow ((\text{Rep } e \text{ as } a, \text{Rep } e \text{ as } b) \rightarrow (\text{Rep } e \text{ as } b, \text{Rep } e \text{ as } a)) \\
\]

\[
\text{lamE} (\text{Proxy :: Proxy True}) \text{ swap} \\
:: (\text{TaglessExp } e, a \in \text{as}, b \in \text{as}) \\
\Rightarrow e \text{ as } ((a, b) \rightarrow (b, a)) \\
\]

Figure 6.5: Using the \text{Rep} type family to type the application of \text{lamE} to the \text{swap} function.
lamE (Proxy :: Proxy True) swap
:: (TaglessExp e, a ∈ as, b ∈ as,
   Rep e as a ∼ (c, d), Rep e as b ∼ (d, c))
⇒ e as (a → b)

6.5.2 Arbitrary-arity functions over algebraic data types

If we are willing to forfeit type inference, some subtle modifications to the instances of the
Variadic class yield an implementation of polyLamE which can embed Rep-based functions of
arbitrary arity. Recall the current definitions:

instance (TaglessExp e, a ∈ as, b ∈ as, f ∼ (a → b))
⇒ Variadic (e as a) (e as b) (e as f) where
polyLamE
= lamE (Proxy :: Proxy True)

instance (TaglessExp e, a ∈ as, Variadic b r s,
   s ∼ e as (c → d), f ∼ (a → e → d))
⇒ Variadic (e as a) (b → r) (e as f) where
polyLamE f
= lamE (Proxy :: Proxy False) (λx → polyLamE (f x))

It turns out that we need not change the implementations of polyLamE; it will be enough to
correct the instance heads and contexts to line up with the new type of lamE. Correcting the
context is not too difficult – given the body of the recursive polyLamE function, for example,
GHC will infer the new instance context for us:

\[ λf → \text{lamE (Proxy :: Proxy False)} (λx → \text{polyLamE} (f x)) \]
:: (TaglessExp e, a ∈ as, Variadic b r s (Rep e as s))
⇒ (Rep e as a → b → r) → e as (a → s)

The head follows through unifying the type to the right of the ⇒ with polyLamE's uninstanti-
ated type \((x → y) → z\) (here α-renamed to avoid name clashes). Note that we must introduce
an additional equality constraint into the instance context as GHC does not allow type family
applications to appear in instance heads. Additionally, we maintain the equality constraints
from the previous instance context to maintain flexibility of instance selection:

instance (TaglessExp e, a ∈ as, Variadic b s t,
   t ∼ Rep e as (c → d), r_a ∼ Rep e as a)
⇒ Variadic r_a (b → s) (e as (a → c → d)) where
polyLamE f
= lamE (Proxy :: Proxy False) (λx → polyLamE (f x))
Note that the expression \((\lambda x \to \text{polyLamE} \,(f \, x))\) in the above definition has the type:

\[
\text{Rep} \, e \, a \to \text{Rep} \, e \, (a \to c \to d)
\]

The instance head states that passing this function to the partial application \(\text{lamE} \,(\text{Proxy} :: \text{Proxy} \, \text{False})\) must yield a value of type \(e \, a \to b \to c\). We thus need one more instance of \(\text{Rep}\) to tie everything together:

\[
\text{type instance} \quad \text{Rep} \, e \, a \to b = e \, a \to b
\]

With this, we may at last type the quotation \([\lambda] \text{swap} \] , e.g.:

\[
[\lambda] \text{swap} \] :: \(\text{Exp}' \, [\text{((Bool, Int), (Int, Bool)}] \,(\text{((Bool, Int) \to (Int, Bool))})
\]

or, using the type quasiquoter also described in the previous section:

\[
[\lambda] \text{swap} \] :: [\lambda] \,(\text{Bool, Int) \to (Int, Bool})
\]

### 6.6 Discussion

In this chapter we have shown how to embed arbitrary Haskell functions into DSL terms using HOAS, demonstrating how optional and generic type class constraints may be integrated to constrain all or part of a function’s type. In doing so we have also explored how polyvariadic functions and quasiquotation can be used to once more hide unnecessary complexity from the DSL user.

While we saw how to recover support for features such as composite types and pattern-matching, we pushed the boundaries of what is possible with Haskell’s type system, finally reaching a point where type inference is no longer possible. Meanwhile, there is still scope for further investigation: the method given in the previous section does not support higher-order functions due to the \(\text{Rep}\) instance for function types. Embedding the map function, for example:

\[
\text{map} :: (a \to b) \to [a] \to [b]
\]

introduces the two equality constraints:

\[
(a \to b) \sim \text{Rep} \, e \, a \to x
\]

\[
([a] \to [b]) \sim \text{Rep} \, e \, a \to y
\]

Unfortunately these two constraints are at odds with either other: the second requires the \(\text{Rep}\) instance we have already defined, allowing it to decompose into the type:
\[ \text{Rep } e \text{ as } a \rightarrow \text{Rep } e \text{ as } b \]

However, this would require map to accept an argument of type \( e \text{ as } (a \rightarrow b) \). The first thus needs a contradictory instance:

**type instance** \( \text{Rep } e \text{ as } (a \rightarrow b) \\
= (a \rightarrow b) \)

We have not yet found a satisfactory solution to this problem. As for the functions that can be embedded, we anticipate that even some of these will be problematic for implemeters of the \( \text{lamE} \) function. Consider a more polymorphic length function, which counts the number of items in a list and returns a result whose type is a member of \( \text{Num} \):

\[
\text{length} :: \text{Num } b \Rightarrow [a] \rightarrow b \\
\text{length } [] = 0 \\
\text{length } (x : xs) = 1 + \text{length } xs
\]

we can embed this function by picking a type in the usual manner:

\[ \lambda | \text{length } ] :: \lambda | [\text{Float} ] \rightarrow \text{Int } ] \]

Recall that this expands to the expression:

\[ \text{Exp } (\text{lamE } (\text{Proxy } :: \text{Proxy True }) (\lambda xs \rightarrow \text{length } xs)) :: \ldots \]

Now, consider the task of generating code which computes the length function, perhaps as part of the C back end developed in Chapter 5. Since \( [] \) is a sum type, the length function has two distinct clauses, each of which must undergo code generation. But how is the platform implementer to know how many times and with what arguments a function \( f \) must be called before he or she can be certain that all code has been generated? One possible solution is to allow the reification of pattern-matching as a first-class construct, perhaps by using a quasiquoter:

\[
\text{length } xs \\
= [\text{match} | \text{xs with} \\
  [] \rightarrow 0 \\
  (y : ys) \rightarrow 1 + \text{length } ys ]
\]

Unfortunately, this is an invasive solution that would require modifying the code of every existing function we wish to lift. Moreover, it is unclear what the quotation would generate that would allow an implementer to observe the patterns matched within a definition. We have thus not explored this or any other solutions in any details, and defer the investigation of such matters to future work.
Haskell’s type system is arguably a product of two systems, one of types and the other of constraints. In this thesis we have shown that, while the constraint system is in many cases too strong or restrictive, the type system is equally strong enough to temper it. In doing so we have presented what is essentially a collection of design-patterns for the type-safe separation of an interface from multiple implementations. Moreover, we have demonstrated that this can be done without sacrificing important features of the host language such as operator overloading, pattern-matching and (in most cases) type inference. The retention of many facets of type inference in particular serves to distinguish our work from implementations which could be developed in full-spectrum dependently-typed languages such as Agda which, although certainly possessing type systems capable of expressing our methods, lack the type inference capabilities that Haskell possesses.

Higher-rank types play an important role in the typing of heterogeneous embedded programs. Unlike solutions such as that of Hughes [25], the generic constraints presented in this thesis interact well with higher-rank types and do not impede a designer’s ability to compose implementations in an elegant and type-safe manner. Additionally, even function types may be flexibly constrained and embedded using compile-time metaprogramming and type-level patterns such as optional class constraints. We conclude therefore that, while the use of generic constraints can lead to verbose definitions and more complicated type signatures, they are a viable method for embedding DSLs whilst minimising the ‘impedance mismatch’ between the type systems of guest and host. Furthermore, we argue that much of the additional verbosity and complexity can be removed through additional tooling, which we discuss briefly in the next section.

7.1 Applications and future work

Section 4.2 hinted that we could add generic constraints to the constructors of a type (or equally, the class methods of a tagless representation) mechanically in many cases. In the case of annotating a GADT, \( \mathcal{T} \), of kind \( \star \rightarrow \star \), for example, a naïve algorithm for doing this might proceed as follows:

- Parameterise all occurrences of \( \mathcal{T} \) by a list of types \( \alpha \) as, giving \( \mathcal{T} \) the new kind \( [\star] \rightarrow \star \rightarrow \star \).
\begin{verbatim}
(deconstrain d) data Exp a where
    ValueE :: a -> Exp a
    AddE   :: Num a => Exp a -> Exp a -> Exp a
    EqE    :: Eq a => Exp a -> Exp a -> Exp Bool
    CondE :: Exp Bool -> Exp a -> Exp a -> Exp a)
\end{verbatim}

Figure 7.1: Deriving generic constraints for Chapter 4's Exp GADT using Template Haskell.

- For each of \( \mathcal{T} \)'s constructors \( C :: \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \mathcal{T} \ as \ a \), add to \( C \)’s type a constraint \( b \in as \) for each \( \tau \) of the form \( \mathcal{T} \ as \ b \).

Figure 7.1 shows how one might introduce generic constraints to the Exp type originally introduced in Chapter 4 by implementing the above algorithm as a Template Haskell (Section 3.7) function deconstrain, of type DecsQ \( \rightarrow \) DecsQ. It is not too difficult to imagine extending the deconstrain function to also constrain the operations of its argument type, in our example perhaps generating a type ExpOp whose constructors mirror those of Exp.

More interesting however is the question of whether or not a platform’s constraints can be derived more intelligently. In the case of the CUDA example from Section 4.5, we might ask whether it is possible to infer the fact that conditional operations should be prohibited, given some first-class description of the platform and the fact that it possesses a SIMD execution model. How such a description might be realised is a topic of interest in its own right: a suitable choice might also enable the derivation of a DSL implementation capable of, for example, modelling the expected performance of a program. Indeed, it does not seem impossible to suggest that platforms might themselves be described using a DSL.

As our work in Section 4.6 demonstrated, a full-spectrum dependent type system such as that possessed by Agda is more than adequate for encoding the techniques we have presented in this thesis. While Agda’s instance arguments (Devriese and Plessens [18]) are designed to provide the support for implicit argument passing that type class overloading offers, they are not as powerful (or dangerous) as Haskell’s type classes. The encoding we present thus passes dictionary arguments explicitly. It is worth exploring whether or not this verbosity could be avoided by using another full-spectrum dependently-typed language such as Idris (Brady and Hammond [9], Brady [8]), which supports type classes proper.

We believe that generic constraints have more applications than have been discussed in this thesis. Whether we are conceptually constraining a list containing types, operations or both, we are in reality always constraining a list of Haskell types. For example, Figure 7.2 adapts an example of Martins et al. [38] to show how generic constraints may be used to safely type context-dependent functions. Here, the function \( f \) makes use of two pieces of context – the locations of the user and his or her home. A context-dependent computation takes place in a
monad \( \mathcal{M} \), which is parameterised by a type \( \textit{cxt} \) which comprises some representation of the context. The goal is to utilise a representation of \( \textit{cxt} \) which statically prevents \( f \) from executing in a context which lacks these two locations. In our case this representation is just another feature set built using \((\in)\) constraints. The \texttt{homeLocation} and \texttt{userLocation} functions expose features representing the locations of the user and his or her home. The type of \( f \), which uses both these functions, will be \textit{inferred} as requiring the union of these two feature sets. In our case, this will be a type of the form:

\[
\begin{align*}
f &:: (\text{Location Home} \in \textit{cxt}, \text{Location User} \in \textit{cxt}) \Rightarrow \ldots \rightarrow \mathcal{M} \textit{cxt} \ldots
\end{align*}
\]

Classes such as \texttt{All} may then be applied as shown throughout this thesis, perhaps, for instance, permitting a run-time implementation to statically require that all the requested locations in a computation are available:

\[
in\text{Context} :: (\text{All (PresentIn cxt}_1 \textit{ cxt}_2)) \Rightarrow \textit{cxt}_1 \rightarrow \mathcal{M} \textit{cxt}_2 a \rightarrow a
\]

Here, the constraint \texttt{PresentIn cxt}_1 \( f \) specifies that the feature \( f \) is present in the context \( \textit{cxt}_1 \). By extending this over a set of features \( \textit{cxt}_2 \) using the \texttt{All} class, we express the requirement that the context \( \textit{cxt}_2 \) is a subset of the context \( \textit{cxt}_1 \). Assuming that the type \( \mathcal{V} \textit{cxt}_1 \) associates each type in the context \( \textit{cxt}_1 \) with a value, then, the \texttt{inContext} function is able to guarantee that any context required by its second argument can be satisfied with an appropriately-typed value.

### 7.2 Closing remarks

Type systems are a vital tool in preventing many classes of programmer error. Domain-specific type systems are arguably more important still as they must ensure the safety of programs written by domain experts who may not be proficient programmers. Embedding a DSL into
a general-purpose programming language should not and need not come at the cost of the strength of the DSL’s type system. In this thesis we have presented generic constraints as a flexible method for embedding domain-specific type systems. While we believe that our techniques help both language designers and consumers to build and use rich heterogeneous DSLs capable of targeting a variety of differently-typed implementations, it is our hope that our contributions will enable future research into generic constraints and their applications.


Philip Wadler. The Expression Problem. Posted to the Java Genericity mailing list and available online at http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (Plain text), November 2012. Last accessed on 03/02/2013.


