Towards Getting Generic Programming Ready for Prime Time

Alexey Luis Rodriguez Yakushev

ISBN 978-90-393-5053-9

Towards Getting Generic Programming Ready for Prime Time

Generiek Programmeren Toegepast in de Praktijk

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof. dr. J. C. Stoof , ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op woensdag 20 mei 2009 des middags te 2.30 uur door

Alexey Luis Rodriguez Yakushev

geboren op 12 september 1979, te Sint-Petersburg, Rusland

Promotoren : Prof. dr. J. Th. Jeuring
Prof. dr. S. D. Swierstra

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

This thesis is based on the following publications and reports:

1. *Generating Generic Functions*, written with Johan Jeuring and Gideon Smeding, and published in the proceedings of the ACM Sigplan Workshop on Generic Programming 2006.

2. *A Lightweight Approach to Datatype-Generic Rewriting*, written with Thomas van Noort, Stefan Holdermans, Johan Jeuring and Bastiaan Heeren, and published in the proceedings of the ACM Sigplan Workshop on Generic Programming 2008.

3. *Generic programming with fixed points for mutually recursive datatypes*, written with Stefan Holdermans, Andres Löh and Johan Jeuring, and published as Technical Report UU-CS-2008-019, Utrecht University, 2008.

4. *Comparing Libraries for Generic Programming in Haskell*, written with Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov and Bruno C. D. S. Oliveira, and published in the proceedings of the ACM SIGPLAN Haskell Symposium 2008.

My contributions to these papers are as follows:

1. I designed and implemented the system described in the paper and did most of the writing of an initial version of the paper. It was revised by Johan Jeuring and me. Gideon Smeding contributed the initial code for the specialization of generic functions in the system.

2. I was heavily involved in writing most of the sections in the paper except for the section on Performance. The initial rewriting code used Generic Haskell in Thomas van Noort's master thesis. The initial formulation of rewriting using type families in Haskell is due to Stefan Holdermans. I reworked the presentation to use standard generic functions and recursion schemes. I also designed the reification technique that allows the specification of rules in terms of the original datatypes. Stefan Holdermans formulated the algorithm for an arbitrary number of meta-variables using type families.

3. The structure types were initially designed by Stefan Holdermans and me. The crucial tagging combinator was proposed by Stefan, while I generalized the design to accommodate multiple datatypes, type-changing recursive occurrences and system-specific representation types. Stefan Holdermans and Andres Löh improved the notation of the initial structure types. The zipper presentation is based on ideas by Stefan, Andres, and myself. I generalized rewriting as presented in the paper above to cover systems of (mutually recursive) datatypes. I was heavily involved in writing most of the paper except the zipper section.

4. I designed and implemented most of the comparison benchmark. I also wrote most of the technical report. Johan Jeuring suggested performing a comparison of the libraries and wrote most of the introduction section and various parts of the paper. The remaining co-authors implemented (sometimes substantial) parts of the benchmark and contributed to various parts of the paper.

Chapter 5 of this thesis is recent work that has not been published before. I independently developed the improved spine view and its application to the generation of well-typed terms.

Contents

1 Introduction

Functional programming languages are increasingly proving to be an effective tool in improving the productivity of programmers. Programmers build solutions faster because they can program in a higher-level of abstraction. Programmers spend less time debugging because functional programming languages guarantee that type errors do not occur during program execution. The two most important features that support these characteristics are higher-order functions and type systems equipped with parametric polymorphism.

Functions are the basic building blocks that enable abstraction. Whenever we identify a commonly occurring piece of code, we encapsulate it as a function and abstract over differences by turning these into a function argument. As functions themselves can be arguments to other functions, differences can be arbitrary computations. Such functions are called higher-order functions, and allow the encapsulation of high-level computational patterns. Functional programs are usually more concise and reliable, because applications of higher-order functions replace redundant program fragments, which are often tedious and error-prone. The function *fold* is a familiar example to functional programmers. Many recursive computations that consume lists can be obtained by passing function arguments to *fold*. Since *fold* already implements the traversal of the list argument, we do not have to spell out the recursion when writing a list consumer. Therefore, it is harder to accidentally write a non-terminating consumer using *fold*.

Basically static type checking guarantees that a function will never be applied to an incompatible argument. For example, the application of boolean negation to a string cannot occur in the execution of a well-typed program. Type checking is especially important with higher-order functions. Without it, we would have to carry out detective work to relate a runtime type error (for example, during the execution of *fold*) to the source location that introduced the problem (an ill-typed *fold* argument). Additionally, modern functional type systems have a form of type abstraction called parametric polymorphism. Functions that use this form of abstraction are called parametrically polymorphic functions. A parametrically polymorphic function can abstract over the type of values on which it performs a computation. The abstracted type becomes an additional argument that must be supplied when the function is applied. For example, *fold* performs a recursive computation over a list, but it is oblivious to the type of the list element. Therefore, the type of list elements is a type argument to *fold*. To use *fold*, we apply it to the type of list elements, and this information is used by the type system to check that the *fold* arguments can handle the element type. The behaviour of a parametrically polymorphic function does not depend on the type argument. For example, *fold* performs the same list traversal regardless of the element type.

Datatype-generic programming increases the power of functional languages even further by allowing the behaviour of functions to depend on type arguments. To distinguish such functions from parametrically polymorphic ones we call them generic functions. Programming with generic functions enables us to define functions that can be reused in a broader range of situations than was previously possible. For example, we can now define a *fold* function the traversal of which proceeds according to the type argument passed: a *fold* on lists would traverse lists and a *fold* on a type describing a tree would traverse values of such a tree type.

This thesis is about making datatype-generic programming easier to apply in practice, thus further improving the usefulness of functional programming languages. Section 1.1 introduces generic programming in more detail, and presents the limitations that we address in this thesis. Section 1.2 presents an overview of the material in the following chapters, which contain our proposed solutions to the above-mentioned limitations of generic programming.

1.1 Datatype-generic programming

The most common technique to define a generic function is to perform an induction on the structure (or shape) of the type argument. The first programming language to support such *type-indexed* definitions was PolyP (Jansson and Jeuring 1997). In PolyP, datatypes are modelled as the application of a fixed-point to a functor. This functor is constructed by means of the *constant*, *product*, *sum* and *identity* functors. If a function is defined for all these cases, it can be applied to any datatype that can be modelled by PolyP.

A PolyP generic function can only be applied to a regular recursive datatype of kind $* \rightarrow *$ such as a list or a binary tree. Therefore PolyP cannot be used with a set of datatype declarations that refer to each other. Such systems of (mutually recursive) datatypes are very useful in practice. For example, they are commonly used in compilers to model abstract syntax trees, where each datatype declaration corresponds to a syntactic category. As a consequence, Generic Haskell was designed as the successor to PolyP, which is based on a different approach to generic programming (Hinze 2000c). In this approach, generic functions are type-indexed values that have kind-indexed types. Now, generic programming can be used on datatypes that have arbitrary kinds, and exhibit more complex forms of recursion (mutual and nested recursion).

Generic Haskell supports a larger set of datatypes than PolyP, but it does not completely subsume the latter. Generic Haskell does not model recursion explicitly, and hence generic functions that explicitly use recursion cannot be defined. For example, the definition of generic *fold* needs access to the recursive positions in datatypes. So generic *fold* can be defined in PolyP but not in Generic Haskell. In general, generic functions are sensitive to the way datatypes are modelled by the generic programming approach. Depending on the particular view on datatypes used, a generic function can become easy or impossible to define. Because there are trade-offs involved in choosing one way to model datatypes over another, it is desirable to have support for different

views on datatypes within a single generic programming approach. It is this motivation that prompted Holdermans et al. (2006) to introduce the *generic views* extension to Generic Haskell.

Type-indexed types are another extension to Generic Haskell. Like a generic function, a type-indexed type is defined by induction on the structure of types. Such definitions are used when a generic function manipulates a data structure that depends on the type over which the function is generic. For example, the *zipper* is a data structure that supports the purely functional navigation of a datatype value. The operations that it supports include descending into a child, moving to the sibling, and navigating back to the parent. The zipper stores the focused subtree and the context in which it occurs. Since the context information is datatype specific, it depends on the datatype being navigated. Because of this dependency the generic variant of the zipper must be defined as a type-indexed type.

The generic views and type-indexed types extensions form an important step in making generic programming more practical. With generic views, functions that previously had to be defined in different generic programming approaches, become definable in one single approach. With type-indexed types, it becomes possible to generalize the definition of a data structure that previously was tied to some specific datatype. However, the usefulness of both extensions can be further improved if some remaining problems are solved:

First of all, even with generic views the fixed-point view is still limited: generic functions can only be applied to regular datatypes. This is a rather unfortunate limitation given the large number of applications that are based on this view. Examples of such applications are the recursion schemes such as *fold* and its variants (Meijer et al. 1991), upwards and downwards accumulations (Bird et al. 1996; Gibbons 1998), unification (Jansson and Jeuring 1997), rewriting and matching functions (Jansson and Jeuring 2000), functions for selecting subexpressions (Steenbergen et al. 2008), pattern matching (Jeuring 1995), etc. Besides these, there are several applications using type-indexed type definitions that are based on the fixed-point view: generic rewriting (Jansson and Jeuring 2000; Van Noort et al. 2008), generic tries, and the zipper and its variants (Huet 1997; McBride 2001; Hinze et al. 2004; Morris et al. 2006; McBride 2008). In summary, generalizing the fixed-point view beyond regular datatypes would make applications that are based on the fixed-point view more useful in practice.

A second problem may arise when a type-indexed type is used in an interface. In PolyP and Generic Haskell, generic functions are defined in terms of type structure, but the arguments passed to these functions are normal datatype values. It follows that knowledge of the type structure is only needed when defining a generic function but not when applying it. This enables the use of libraries that internally apply generic programming techniques by programmers that have no knowledge of generic programming. However, when a generic function has an argument with a type-indexed type, constructing such an argument requires detailed knowledge regarding type structure and the type-indexed type definition. While sometimes it is possible to define generic helper functions that shield the programmer from these details, there are examples where this solution is extremely cumbersome. Generic rewriting is an application

where this problem occurs. The type of rewrite rules is defined in terms of a type-indexed type. Therefore, specifying a rewrite rule requires either an extensive invocation of helper functions or detailed knowledge of type structure and the type-indexed type definition. In summary, libraries using type-indexed types, might require the user of the library to become familiar with implementation details of it.

So far, we have discussed issues regarding expressiveness (generality of the fixed-point view) and convenience (how to construct values of a type-indexed type). Although our discussion centered around the Generic Haskell and PolyP programming languages, the question may arise as to whether we really need such a special purpose language to do datatype generic programming. The answer is probably negative. It is possible to do generic programming directly in Haskell, and in fact there are a number of approaches that already do so. It is possible to implement libraries that support a Generic Haskell style (Cheney and Hinze 2002; Oliveira et al. 2006), a PolyP style (Norell and Jansson 2004), and other styles of generic programming (Lämmel and Peyton Jones 2003; Mitchell and Runciman 2007b).

Being able to do generic programming directly in Haskell reduces the barrier to use generic programming: it is often possible to just download a generic programming library, import it and start using generic programming in a project. However, there are significant differences between all Haskell-based approaches. For example, they can use different views on datatypes or different Haskell features (type classes, rank-n polymorphism). Because these choices affect expressiveness and convenience of use, it is not easy to tell which approach is to be preferred when writing a generic function. There is clearly a need for a characterisation of the available generic programming approaches for Haskell.

Haskell is an evolving language. Features that increase its expressiveness are constantly being proposed and implemented. Some of these features make datatype declarations more powerful, so it is natural to ask whether generic programming can be used on those improved datatypes. Since their introduction to Haskell, Generalized Algebraic Datatypes (GADTs) (Xi et al. 2003; Cheney and Hinze 2003; Peyton Jones et al. 2006) have often been applied to improve the reliability of programs. GADTs allow the encoding of datatype invariants by type constraints in constructor signatures. With this information, the compiler rejects during the type-checking process values for which such invariants do not hold. In particular, GADTs can be used to model sets of well-typed terms such that values representing ill-typed terms cannot be constructed.

Given the growing relevance of GADTs, it is important to provide generic programming support for generalized datatype definitions. The automatic generation of datatype values for the purpose of software testing is of particular interest. The generation of values based on normal datatype definitions is often too imprecise: it gives rise to either values that do not occur in practice, or, worse, ill-formed values (for example, a program fragment with unbound variables). Since GADTs are more precise than normal datatypes, there is the question of whether applying generic programming to GADTs can produce values that are better suited for testing program properties. For example, it should be possible to use a generic value generation function with a GADT encoding lambda calculus, in order to produce a well-typed lambda term and test with

it a tag-less interpreter.

1.2 Contributions and overview

The contributions of this thesis are:

- In Chapter 2, we present a complete example of generic programming based on the fixed-point view and type-indexed types. We do not use Generic Haskell and PolyP, but instead, we write the example directly in Haskell, using the recent language extension of type families. Our example is a reusable rewriting library that is generic in the datatype to be transformed. In generic rewriting, type-indexed types are used to extend any regular datatype with a meta-variable alternative. This additional alternative is necessary to specify rewrite rules. Once this extension is performed, rewriting becomes a generic programming exercise on type-indexed types.

- In Chapter 2, we also show how to build values of a type-indexed type without exposing the user of generic rewriting to the structure of types. Our solution enables the user to specify rewrite rules in terms of the constructors of the original datatype. Therefore the user is not required to understand the fine points of datatype extension with metavariables. Although our solution is specific to generic rewriting, we claim that it can be used in other type-indexed type applications.

- In Chapter 3, we show how to generalize the fixed point view on datatypes so it can handle (mutually recursive) systems of datatypes. This technique generalizes the rewriting example from Chapter 2 so that it can be applied to, for example, abstract syntax trees described by mutually recursive datatypes. This technique also generalizes much of the work on generic programming for fixed-points of functors (Meijer et al. 1991; Bird et al. 1996; Jansson and Jeuring 1997; McBride 2001; Hinze et al. 2004). We illustrate this increased generality by presenting a few selected examples: recursion schemes such as folding and compos, the zipper and generic matching.

- In Chapter 4, we characterise and compare many of the generic programming approaches available for Haskell. We introduce a set of criteria and design a generic programming benchmark: a set of characteristic generic programming examples that test criteria compliance. This comparison is indispensable for someone who has to choose the right approach for a project, but also may serve as a starting point for designing new libraries.

- In Chapter 5, we address the problem of generating well-typed terms using generic programming. Well-typed terms are encoded by generalized algebraic datatypes (GADTs) and often also with existential types. The Spine approach (Hinze et al. 2006; Hinze and Löh 2006) to generic programming supports GADTs, but unfortunately it does not support the definition of generic producers for existentials.

We describe how to extend the Spine approach to support existentials and we use the improved Spine to define a generic enumeration function. We show that the enumeration function can be used to generate the terms of simply typed lambda calculus.

In the list of contributions above, we have addressed a number of issues hoping that generic programming becomes more applicable and accessible to the Haskell programmer. In the same spirit, we would also like to have tools that offer support to the beginning generic programmer. In the functional programming world there exist tools that generate programs from a type. For example, such a tool would generate the code for function composition when presented with its type. A similar tool could help the beginning generic programmer with taking the "abstraction leap" from a non-generic function to a generic one. For example, the programmer could present the hypothetical tool with an instance of a generic function (say *map* on lists) and the type that the generic function should have, then the tool would produce the code for generic map. Our contribution is a small step towards that goal:

- In Chapter 6, we present a system that can synthesize the code for a generic function from a specification consisting of a type, type-specific instances of the generic function, and the properties that the function should satisfy. This work builds on work by Augustsson (2005), which uses the Curry-Howard isomorphism and automatic theorem proving techniques (Dyckhoff 1992) to generate well-typed lambda calculus terms.

2 A Lightweight Approach to Datatype-generic Rewriting

This chapter presents generic rewriting, a complete example of generic programming using the fixed-point view on datatypes and type-indexed type definitions. This presentation uses Haskell extended with type families.

Previous implementations of generic rewriting libraries have a number of limitations: they require the user to either adapt the datatype on which rewriting is applied, or to specify the rewriting rules as functions; the latter makes it hard or impossible to document, test, and analyse rules. We describe a library that demonstrates how to overcome these limitations by defining rules in terms of datatypes, and show how to use a type-indexed datatype to automatically extend a datatype for syntax trees with a case for metavariables. We then show how rewrite rules can be implemented without any knowledge of how the datatype is extended with metavariables. We analyse the performance of our library and compare it with other approaches to generic rewriting.

2.1 Introduction

Consider a Haskell datatype Prop for representing expressions of propositional logic:

$$\begin{aligned}&\textbf{data}\ \mathsf{Prop} = \mathit{Var}\ \mathsf{String} \mid T \mid F \mid \mathit{Not}\ \mathsf{Prop}\\ &\qquad\qquad \mid\ \mathsf{Prop} :\wedge: \mathsf{Prop} \mid \mathsf{Prop} :\vee: \mathsf{Prop}\end{aligned}$$

Now suppose we wish to simplify such expressions using the law of contradiction: $p \wedge \neg p \rightarrow F$. Ideally our formulation of this rewrite rule as an executable program should neither be much longer nor much more complicated than this rule itself. One approach is to encode this rule as a function and to apply it to an expression using a bottom-up traversal function *transform*:

$$\begin{aligned}&\mathit{simplify} :: \mathsf{Prop} \rightarrow \mathsf{Prop}\\ &\mathit{simplify}\ \mathit{prop} = \mathit{transform}\ \mathit{andContr}\ \mathit{prop}\\ &\quad \textbf{where}\\ &\qquad \mathit{andContr}\ (p :\wedge: \mathit{Not}\ q) \mid p \equiv q = F\\ &\qquad \mathit{andContr}\ p \qquad\qquad\qquad\quad = p\end{aligned}$$

Although this definition is relatively straightforward, encoding rules by functions has a number of drawbacks. To start with, a rule is not a concise one-line definition, because we have to provide a catch-all case in order to avoid pattern-matching failure at runtime.

Second, pattern guards are needed to deal with multiple occurrences of a variable, cluttering the definition. Lastly, rules cannot be analysed easily since it is hard to inspect functions.

One way to solve these drawbacks is to design specialized rewriting functionality. We could define a datatype representing rewriting rules on propositions, and implement associated rewriting machinery that supports patterns with metavariable occurrences. While the drawbacks mentioned above are solved, this solution has a serious disadvantage: it entails a large amount of datatype-specific code. If we wanted to use rewriting on, say, a datatype representing arithmetic expressions, we would have to define new rewriting functions and a new datatype for rules.

In this chapter we present a rewriting library that is generic in the datatype to which rules are applied. Using our library, the example above can be written as:

$$
\begin{aligned}
&\mathit{simplify} :: \mathsf{Prop} \to \mathsf{Prop} \\
&\mathit{simplify}\ \mathit{prop} = \mathit{transform}\ (\mathit{applyRule}\ \mathit{andContr})\ \mathit{prop} \\
&\quad \textbf{where} \\
&\qquad \mathit{andContr}\ p = p :\wedge: \mathit{Not}\ p :\rightsquigarrow F
\end{aligned}
$$

The library provides *transform*, *applyRule*, and $(:\rightsquigarrow)$ which are generic and in this case, instantiated with the type Prop. There is no constructor for metavariables. Instead, we use ordinary function abstraction to make the metavariable p explicit in the rule *andContr*, which is now a direct translation of the rule $p \wedge \neg p \to F$. This rule description no longer suffers from the drawbacks of the solution that was based on pattern matching.

Specifically, these are the contributions of our chapter:

- Our library implements term rewriting using generic programming techniques within Haskell extended with associated type synonyms (Chakravarty et al. 2005), as implemented in the Glasgow Haskell Compiler (GHC). In order to specify rewrite rules, terms have to be extended with a constructor for metavariables. This extension is achieved by making the datatype representing rules type-indexed (Hinze et al. 2004). The rewriting machinery itself is implemented using well-known generic functions such as *map*, *crush*, and *zip*.

- We present a new technique to specify holes in a tree without the need to modify the datatype representing trees. We use it to make it easy for a user to write rules using our library: rules are defined using the original term constructors. This is remarkable because the values of type-indexed datatypes (which in our library are used to represent terms extended with metavariables) must be built from a different set of constructors than the set normally used in the program (the constructors of Prop, in our example). Thus, we hide details about the use of type-indexed types from the user who writes rules. This technique can also be applied to other generic programs. In programs using a generic zipper data structure (Hinze et al. 2004), it could be used to conveniently describe an initial zipper value.

Besides these contributions, our rewriting library is an elegant example of how to use associated types to implement type-indexed types in a lightweight fashion. Most examples of type-indexed types (Hinze et al. 2004; Van Noort 2008) have been implemented using a language extension such as Generic Haskell (Löh 2004). Other examples of lightweight implementations of type-indexed types have been given by Chakravarty et al. (2008), and Oliveira and Gibbons (2005).

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

This chapter is organized as follows. Section 2.2 continues the discussion on how to represent rewrite rules and motivates the design choices we made in our library. In Section 2.3, we present the interface of our library followed (Section 2.4) by a detailed description of its implementation and the underlying machinery. We compare the efficiency or our library to other approaches to term rewriting in Section 2.5. Finally, we discuss related work in Section 2.6, and we present our conclusions and ongoing research in Section 2.7.

2.2 Motivation

There are at least two techniques to implement rule-based rewriting in Haskell: "rules based on pattern matching" and "rules as values of datatypes".

2.2.1 Rules Based on Pattern Matching

The first technique encodes rewrite rules as Haskell functions, using pattern matching to check whether the argument term matches the left-hand side of the rule. If this is the case, then we return the right-hand side of the rule, or else, we return the term unchanged. For example, the rule $\neg(p \wedge q) \rightarrow \neg p \vee \neg q$ is implemented as follows:

$$
\begin{aligned}
&\mathit{deMorgan} :: \mathsf{Prop} \rightarrow \mathsf{Prop} \\
&\mathit{deMorgan}\ (\mathit{Not}\ (p :\wedge: q)) = \mathit{Not}\ p :\vee: \mathit{Not}\ q \\
&\mathit{deMorgan}\ p \qquad\qquad\quad = p
\end{aligned}
$$

We have to add a catch-all case, because otherwise a runtime failure is caused by an argument that does not match the pattern.

Rules containing variables with multiple occurrences in the left-hand side cannot be encoded as Haskell functions directly. Instead, such occurrences must be enforced in so-called pattern guards. For example, $p \vee \neg p \rightarrow T$ is implemented by:

$$\begin{aligned} &\mathit{exclMiddle} :: \mathsf{Prop} \rightarrow \mathsf{Prop} \\ &\mathit{exclMiddle}\ (p :\vee: \mathit{Not}\ q) \mid p \equiv q = T \\ &\mathit{exclMiddle}\ p \qquad\qquad\qquad\quad = p \end{aligned}$$

Here we have replaced the second occurrence of p with a fresh variable q, but we have retained the equality constraint as a pattern guard ($p \equiv q$). Note that this requires the availability of an equality function for the type Prop.

In some situations, it is useful to know whether a rule has been successfully applied or not. We provide this information by returning the rewriting result in a Maybe value, at the expense of some additional notational overhead:

$$\begin{aligned} &\mathit{exclMiddleM} :: \mathsf{Prop} \rightarrow \mathsf{Maybe\ Prop} \\ &\mathit{exclMiddleM}\ (p :\vee: \mathit{Not}\ q) \mid p \equiv q = \mathit{Just}\ T \\ &\mathit{exclMiddleM}\ p \qquad\qquad\qquad\quad = \mathit{Nothing} \end{aligned}$$

Encoding rules by functions is very convenient because we can directly use generic traversal combinators that are parametrized by functions. For example, the Uniplate library (Mitchell and Runciman 2007b) defines *transform*,

$$\mathit{transform} :: \mathsf{Uniplate\ a} \Rightarrow (\mathsf{a} \rightarrow \mathsf{a}) \rightarrow \mathsf{a} \rightarrow \mathsf{a}$$

which applies its argument in a bottom-up fashion, together with many other traversal combinators.

This combinator can be used to remove some tautological statements from a proposition:

$$\begin{aligned} &\mathit{removeTaut} :: \mathsf{Prop} \rightarrow \mathsf{Prop} \\ &\mathit{removeTaut} = \mathit{transform}\ \mathit{exclMiddle} \end{aligned}$$

Pattern matching allows for rewrite rules to be encoded more or less directly as functions. Furthermore, on account of their functional nature, rules can be immediately combined with generic programming strategies in order to control their application. However, having rules as functions raises a number of issues:

- The rules cannot be observed easily because it is not possible to directly inspect a function in Haskell. There are several reasons why it is desirable to observe rules:
  - Documentation: The rules of a rewriting system can be pretty-printed to generate documentation.

  - Automated testing: In general, a rule must preserve the semantics of the expression that it rewrites. A way to test this property is to randomly generate terms and check whether the rewritten version has the same semantics. However, a rule with a complex left-hand side will most likely not match a randomly generated term, and hence it will not be tested sufficiently. If the left-hand side of a rule can be inspected we can direct the generation process in order to improve testing coverage.
  - Inversion: The left-hand side and right-hand side of a rule can be exchanged, resulting in the inverse of that rule.
  - Tracing: When a sequence of rewriting steps leads to an unexpected result, one may want to learn which rules were applied in which order.

- The lack of nonlinear pattern matching in Haskell can become a nuisance if the left-hand side of a rule contains many occurrences of the same variable.
- It is tedious to have to specify a catch-all case when rules are encoded as functions. All rule definitions require this extra case.
- Haskell is not equipped with first-class pattern matching, so the user cannot write abstractions for commonly occurring structures in the left-hand side of a rule.

2.2.2 Rules as Values of a Datatype

Instead of using functions, we use a datatype to encode our rewrite rules, which makes left-hand sides and right-hand sides observable:

$$\textbf{data}\ \mathsf{RuleSpec\ a} = \mathsf{a} :\rightsquigarrow \mathsf{a}$$

In the case of our example, we have to introduce a variation of the Prop datatype, which has an extra constructor *MetaVar* representing metavariables. This extended datatype EProp is used to define rewrite rules.

$$\begin{aligned}\textbf{data}\ \mathsf{EProp} = {} & \mathit{MetaVar}\ \mathsf{String} \\ \mid {} & \mathit{EVar}\ \mathsf{String} \mid \mathit{ET} \mid \mathit{EF} \mid \mathit{ENot}\ \mathsf{EProp} \\ \mid {} & \mathsf{EProp} :\triangle: \mathsf{EProp} \mid \mathsf{EProp} :\triangledown: \mathsf{EProp}\end{aligned}$$

Of course, we need to define a rewriting function specific to propositions that uses rules expressed by value of EProp to transform expressions:

$$\mathit{rewriteProp} :: \mathsf{RuleSpec\ EProp} \rightarrow \mathsf{Prop} \rightarrow \mathsf{Prop}$$

Here we do not show the implementation of *rewriteProp*, but note that its type (and thus its implementation) is specific to propositions. If we want to specify rewrite rules that work on a different datatype, then we also have to define a new *rewrite* function for it. With the specific rewrite function for propositions we can use rules on propositions as in:

```
   -- Rule specifications and rules (ADT)
data RuleSpec a = a :⇝ a
type Rule a


   -- Build rules from specifications with 0, 1, and 2 metavariables
rule0 :: Rewrite a ⇒                RuleSpec a   → Rule a
rule1 :: Rewrite a ⇒ (       a → RuleSpec a) → Rule a
rule2 :: Rewrite a ⇒ (a → a → RuleSpec a) → Rule a


   -- Build rules from specifications with any number of metavariables
rule :: (Builder r, Rewrite (Target r)) ⇒ r → Rule (Target r)


   -- Application of rules to terms
rewriteM :: (Rewrite a, Monad m) ⇒ Rule a → a → m a
rewrite  ::  Rewrite a           ⇒ Rule a → a → a


   -- Application of rule specifications to terms
applyRule :: (Builder r, Rewrite (Target r))
          ⇒ r → Target r → Target r
```

Figure 2.1: Generic rewriting library interface

```
simplifyProp :: Prop → Prop
simplifyProp = transform (rewriteProp exclMiddle)
  where
    exclMiddle =
      MetaVar "p" :⍰: ENot (MetaVar "p") :⇝ ET
```

In this definition, another problem becomes apparent. The way in which rules are specified is very inconvenient, because, rather than re-using the type Prop, we are forced to use the (associated) new datatype EProp with its new set of constructor names.

2.3 Interface

The interface to our generic rewriting library is presented in Figure 2.1. We explain it by means of a series of examples. We first specify a few rules to be used with the library:

```
notTrue    ::                          RuleSpec Prop
andContr   ::                 Prop → RuleSpec Prop
deMorgan   :: Prop → Prop → RuleSpec Prop

notTrue          = Not T          :⇝ F
andContr  p      = p :∧: Not p    :⇝ F
deMorgan p q     = Not (p :∧: q)  :⇝ Not p :∨: Not q
```

Rule specifications represent metavariables as function arguments. For example, the *deMorgan* rule uses two metavariables, so they are introduced as function arguments p and q. Rules with no metavariables, such as *notTrue*, do not take any argument.

Before we can use rules for rewriting, we transform them first to an internal representation. Let us take a look at the functions $rule_0$, $rule_1$, and $rule_2$. Each takes a rule specification and returns a rule. Each function handles rules with a specific number of metavariables. In principle, the user would have to pick the right function to build a rule, but in practice, it is more convenient to abstract over the number of metavariables by means of a type class. The library provides an overloaded function *rule* for building rules from specifications with any number of metavariables[1]:

```
notTrueRule, andContrRule, deMorganRule :: Rule Prop

notTrueRule    = rule notTrue
andContrRule   = rule andContr
deMorganRule   = rule deMorgan
```

The library defines two rewriting functions, namely *rewriteM* and *rewrite*. The first has a more informative type: the monad is used to notify about failure if the term does not match against the rule. The second rewriting function always succeeds, and returns the term unchanged whenever the rule is not applicable. Consider the following expressions:

```
rewrite   notTrueRule   (Not T)
rewrite   andContrRule  (Var "x" :∧: Not (Var "x"))
rewriteM deMorganRule (T :∧: F)
```

These expressions evaluate to F, F, and *Nothing* respectively. Sometimes, it is more practical to directly apply a rule specification, without calling the intermediate transformation step (*rule*). The function *applyRule* can be used for this purpose:

```
applyRule deMorgan (Not (T :∧: Var "x"))
```

This expressions evaluates to `Not T :∨: Not (Var "x")`.

2.3.1 Representing the Structure of Datatypes

To enable generic rewriting on a datatype, a user must describe the structure of that datatype to the library. This is a process analogous to that of Scrap Your Boilerplate

[1]The associated type synonym `Target` returns the datatype to which the rule is applied. For example, `Target (Prop → RuleSpec Prop)` yields `Prop`. For more details, see Section 2.4.4.

```
class Functor (PF a) ⇒ Regular a where
  type PF a :: ∗ → ∗
  from      :: a      → PF a a
  to        :: PF a a → a

data K a r      = K a
data Id r       = Id r
data Unit r     = Unit
data (f :+: g) r = Inl (f r) | Inr (g r)
data (f :×: g) r = f r :×: g r

infixr 7 :×:
infixr 6 :+:
```

Figure 2.2: Regular type class and view types

(Lämmel and Peyton Jones 2003) and Uniplate. These libraries require datatypes to be instances of the classes Data and Uniplate respectively.

In our library, the structure of a regular datatype is given by an instance of the type class Regular. Figure 2.2 shows the definition of Regular and the type constructors used to describe type structure. This type class declares PF, an associated type synonym of kind $* \to * \to *$ that abstracts over the immediate subtrees of a datatype and captures the notion of *pattern functor*. For example, Figure 2.3 shows the complete definition of the Regular instance for Prop. The associated type synonym PF corresponds to PolyP's type constructor FunctorOf (Jansson and Jeuring 1997; Norell and Jansson 2004). Its instantiation follows directly from the definition of a datatype. Choice amongst constructors is encoded by nested sum types (:+:); therefore, five sums are used above to encode six constructors. A constructor with no arguments is encoded by Unit; this is the case for the second and third constructors (*T* and *F*). Constructor arguments that are recursive occurrences (such as that for *Not*) are encoded by Id. Other constructor arguments are encoded by K, and, hence, the argument of *Var* is encoded in that way. Finally, constructors with more than a single argument (like (:∧:) and (:∨:)) are encoded by nested product types (:×:).

The two class methods of Regular, *from* and *to*, relate datatype values with their structural representation. Together, they establish a so-called *embedding-projection pair*: i.e., they should satisfy $to \circ from = id$ and $from \circ to \sqsubseteq id$ (Hinze 2000a). More specifically, *from* transforms the top-level constructor into a structure value, while leaving the immediate subtrees unchanged. The function *to* performs the transformation in the opposite direction.

We emphasize that although the instance declaration for Prop may seem quite verbose, it follows directly and mechanically from the structure of the datatype. Defining instances of Regular could easily be done automatically, for example by using Template Haskell (Sheard and Peyton Jones 2002). Moreover, all that needs to be done to

```
instance Regular Prop where
  type PF Prop = K String :+: Unit :+: Unit :+: Id :+: Id :×: Id :+: Id :×: Id
  from (Var s)   = Inl (K s)
  from T         = Inr (Inl Unit)
  from F         = Inr (Inr (Inl Unit))
  from (Not p)   = Inr (Inr (Inr (Inl (Id p))))
  from (p :∧: q) = Inr (Inr (Inr (Inl (Id p :×: Id q)))))
  from (p :∨: q) = Inr (Inr (Inr (Inr (Id p :×: Id q)))))
  to (Inl (K s))                                  = (Var s)
  to (Inr (Inl Unit))                             = T
  to (Inr (Inr (Inl Unit)))                       = F
  to (Inr (Inr (Inr (Inl (Id p)))))               = (Not p)
  to (Inr (Inr (Inr (Inr (Inl (Id p :×: Id q)))))) = (p :∧: q)
  to (Inr (Inr (Inr (Inr (Id p :×: Id q)))))) = (p :∨: q)
```

Figure 2.3: Complete Regular instance for Prop

use the generic rewriting library on a datatype is declaring it as an instance of Regular and Rewrite (see Section 2.4.5).

2.4 Implementation

In this section we describe the implementation of the library. We proceed as follows. First, we explain how generic functionality is implemented for datatypes that are instances of Regular. In particular, we explain how to implement *map*, *crush*, and *zip* generically over pattern functors. Next, we present the implementation of generic rewriting using these functions. Finally, we describe how to support rule specifications that use the original datatype constructors.

2.4.1 Generic Functions

Generic functions are defined once and can be used on any datatype that is an instance of Regular. The definition of a generic function is given by induction on the types used to describe the structure of a pattern functor. In our library, generic function definitions are given using type classes.

Generic Map

Mapping over the elements of a pattern functor is defined by means of the Functor type class, as given in Figure 2.4. Here, the interesting case is the transformation of pattern functor elements, which are stored in the *Id* constructors. Note that this is a

```
class Functor f where
  fmap :: (a → b) → f a → f b
instance Functor Id where
  fmap f (Id r) = Id (f r)
instance Functor (K a) where
  fmap _ (K x) = K x
instance Functor Unit where
  fmap _ Unit = Unit
instance (Functor f, Functor g) ⇒ Functor (f :+: g) where
  fmap f (Inl x) = Inl (fmap f x)
  fmap f (Inr y) = Inr (fmap f y)
instance (Functor f, Functor g) ⇒ Functor (f :×: g) where
  fmap f (x :×: y) = fmap f x :×: fmap f y
```

Figure 2.4: *fmap* definitions

well-known way to implement generic functions, derived from how generic functions are implemented in PolyP.

Now, using *fmap*, we define the *compos* operator of Bringert and Ranta (2006), which applies a function to the immediate subtrees of a value. A pattern functor abstracts precisely over those subtrees, so we use Regular to define *compos* generically:

```
compos :: Regular a ⇒ (a → a) → a → a
compos f = to ∘ fmap f ∘ from
```

The function *compos* transforms the a argument into a value of the pattern functor, so that it can apply *f* to the immediate subtrees using *fmap*. The version of *compos* given here is equivalent to PolyP's *mapChildren* (Jansson and Jeuring 1998a); we could also define monadic or applicative variants of this operator by generalizing generic mapping in a similar fashion.

Generic *compos* can be used, for example, to implement the bottom-up traversal *transform*, used in the introduction:

```
transform :: Regular a ⇒ (a → a) → a → a
transform f = f ∘ compos (transform f)
```

Generic Crush

A *crush* is a useful fold-like operation on pattern functors. It combines all the a values within pattern-functor value into a b-value using a binary operator $(\oplus)$ and an initial b-value e. The definition of *crush* is given in Figure 2.5.

```
class Crush f where
  crush :: (a → b → b) → b → f a → b
instance Crush Id where
  crush (⊕) e (Id x) = x ⊕ e
instance Crush (K a) where
  crush _ e _ = e
instance Crush Unit where
  crush _ e _ = e
instance (Crush a, Crush b) ⇒ Crush (a :+: b) where
  crush (⊕) e (Inl x) = crush (⊕) e x
  crush (⊕) e (Inr y) = crush (⊕) e y
instance (Crush a, Crush b) ⇒ Crush (a :×: b) where
  crush (⊕) e (x :×: y) = crush (⊕) (crush (⊕) e y) x
```

Figure 2.5: *crush* definitions

A common use of *crush* is collecting all recursive elements of a pattern-functor value:

```
flatten :: Crush f ⇒ f a → [a]
flatten = crush (:) []
```

If a datatype is an instance of Regular, we obtain the immediate recursive occurrences of a value by means of *children*:

```
children :: (Regular a, Crush (PF a)) ⇒ a → [a]
children = flatten ∘ from
```

Generic Zip

Generic *zip* over pattern functors is crucial to the implementation of rewriting. Figure 2.6 shows the definition of generic *zip* over pattern functors, *fzipM*, which takes a function that combines the a and b values that are stored in the pattern functor structures. It traverses both structures in parallel and merges all occurrences of a and b values along the way. The function has a monadic type because the structures may not match in the case of sums (which corresponds to different constructors) and constant types (because of different values stored in *K*). However, the monadic type also allows the merging function to fail or to use state, for example.

There are some useful variants of *fzipM*, such as a zip that uses a non-monadic merging function:

```
fzip :: (Zip f, Monad m) ⇒ (a → b → c) → f a → f b → m (f c)
fzip f = fzipM (λx y → return (f x y))
```

```
class Zip f where
  fzipM :: Monad m ⇒ (a → b → m c) → f a → f b → m (f c)
instance Zip Id where
  fzipM f (Id x) (Id y) = liftM Id (f x y)
instance Eq a ⇒ Zip (K a) where
  fzipM _ (K x) (K y)
    | x ≡ y      = return (K x)
    | otherwise = fail "fzipM: structure mismatch"
instance Zip Unit where
  fzipM _ Unit Unit = return Unit
instance (Zip a, Zip b) ⇒ Zip (a :+: b) where
  fzipM f (Inl x) (Inl y) = liftM Inl (fzipM f x y)
  fzipM f (Inr x) (Inr y) = liftM Inr (fzipM f x y)
  fzipM _ _ _ = fail "fzipM: structure mismatch"
instance (Zip a, Zip b) ⇒ Zip (a :×: b) where
  fzipM f (x1 :×: y1) (x2 :×: y2) =
    liftM2 (:×:) (fzipM f x1 x2) (fzipM f y1 y2)
```

Figure 2.6: *fzipM* definitions

and a partial generic zip that does not have a monadic return type:

```
fzip' :: Zip f ⇒ (a → b → c) → f a → f b → f c
fzip' f x y =
  case fzip f x y of
    Just res → res
    Nothing → error "fzip': structure mismatch"
```

2.4.2 Generic Rewriting

Now that we have defined basic generic functions, we continue with defining the basic rewriting machinery. Rules consist of a left-hand side and a right-hand side which are combined using the infix constructor (:⇝):

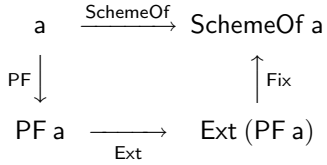
```
data RuleSpec a = a :⇝ a
lhs, rhs :: RuleSpec a → a
lhs (x :⇝ _) = x
rhs (_ :⇝ y) = y
```

Schemes

What is the type of rules that rewrite Prop-values? The type RuleSpec Prop is a poor choice, because such rules cannot contain metavariables. Our solution is to define a type that adds a metavariable case to a datatype. For this purpose, we define a type-indexed type SchemeOf for *schemes*. A scheme extends its argument with a metavariable constructor. Now, the type of rules that rewrite Prop-values is RuleSpec (SchemeOf Prop). We define a type synonym Rule such that this type can be written more concisely as Rule Prop:

```
type Rule a = RuleSpec (SchemeOf a)
```

The process of obtaining SchemeOf a for a given type a can be depicted as follows:



To define SchemeOf a, we assume that a is an instance of Regular and, hence, has an associated pattern functor PF a. Furthermore, we extend the pattern functor with a case for metavariables. The type of schemes of a is then defined as the fixed point of the now extended pattern functor.

To extend a pattern functor with a case for metavariables, it is, in essence, enough to introduce a sum to encode the choice between the extra metavariable case and a value from the original pattern functor:

```
type Ext f = K MetaVar :+: f
```

However, Ext extends a type with metavariables on the top-level only, whereas we also have to allow metavariables to occur in subterms. In other words, the pattern functor has to be extended *recursively*. To this end, we introduce a type synonym Scheme that encodes the recursive structure of schemes by means of a type-level fixed-point operator Fix:

```
newtype Fix f  = In{out :: f (Fix f)}
type Scheme f = Fix (Ext f)
```

A scheme for a given regular type is now defined in terms of the type-indexed type PF:

```
type SchemeOf a = Scheme (PF a)
```

Our use of type-indexed types is simpler than in other applications such as the Zipper, because there are no types defined by induction on the pattern functor.

It remains to define the type MetaVar. In our library, metavariables are defined by Int-values:

```
type MetaVar = Int
```

The choice for Int is rather arbitrary, and other representations could be used as well. In Sections 2.4.3 and 2.4.4 we show that the use of MetaVar-values can be made internal to the library and that the concrete representation of metavariables need not be exposed to the user.

To easily construct Scheme-values we define two helper functions, which construct a metavariable, or a pattern functor value, respectively.

```
metaVar :: MetaVar → Scheme f
metaVar = In ∘ Inl ∘ K
pf :: f (Scheme f) → Scheme f
pf = In ∘ Inr
```

Rewriting functions are often defined by case analysis on values of type Scheme. Therefore, we use a view on Scheme to conveniently distinguish metavariables from pattern functor values.

```
data SchemeView f = MetaVar MetaVar | PF (f (Scheme f))
schemeView :: Scheme f → SchemeView f
schemeView (In (Inl (K x))) = MetaVar x
schemeView (In (Inr r))     = PF r
```

We also define a function to embed a-values into their extended counterparts.

```
toScheme :: Regular a ⇒ a → SchemeOf a
toScheme = pf ∘ fmap toScheme ∘ from
```

Finally, we define a fold on Scheme values that applies its argument functions to either a metavariable or a pattern functor value.

```
foldScheme :: Functor f ⇒ (MetaVar → a) → (f a → a) → Scheme f → a
foldScheme f g scheme =
  case schemeView scheme of
    MetaVar x → f x
    PF r      → g (fmap (foldScheme f g) r)
```

Basic Rewriting

When rewriting a term with a given rule, the left-hand side of the rule is matched to the term, resulting in a substitution mapping metavariables to terms:

```
type Subst a = Map MetaVar (a, SchemeOf a)
```

We store both the original term a and the corresponding scheme SchemeOf a for efficiency reasons. This approach prevents the matched subterm from being converted

multiple times since the right-hand side of a rule can contain multiple occurrences of a metavariable. Instead, each occurrence is instantiated just by selecting the second component from the substitution.

A substitution is obtained by the function *match*, which is passed a scheme and a value of the original type:

```
match :: (Regular a, Crush (PF a), Zip (PF a), Monad m)
      ⇒ SchemeOf a → a → m (Subst a)
match scheme term =
  case schemeView scheme of
    MetaVar x → return (singleton x (term, toScheme term))
    PF r      → fzip (,) r (from term) >>=
                  crush matchOne (return empty)
  where
    matchOne (term1, term2) msubst =
      do subst1 ← msubst
         subst2 ← match (apply subst1 term1) term2
         return (union subst1 subst2)
```

When we encounter a metavariable we return the singleton substitution mapping the metavariable to the original term and its representation as a scheme. Otherwise, we zip the two terms to check that the structures match. Then, we obtain a single substitution by matching each recursive occurrence and returning the union of the resulting substitutions. In the definition of *matchOne*, we apply the substitution obtained thus far to the first term using the function *apply*. This function enforces linear patterns by instantiating each metavariable for which there is a binding. This also guarantees that the resulting substitution does not overlap with the substitution obtained thus far, since multiple occurrences of a metavariable are replaced throughout the term. As a consequence, we can just return the union of the two substitutions.

The function *apply* is defined in terms of *foldScheme*:

```
apply :: Regular a ⇒ Subst a → SchemeOf a → SchemeOf a
apply subst = foldScheme findMetaVar pf
  where
    findMetaVar x = maybe (metaVar x) snd (lookup x subst)
```

When a metavariable is encountered, we lookup the corresponding term in the second component of the substitution. A pattern functor value is reconstructed using the function *pf*.

The function *inst*, which is similar to *apply*, instantiates each metavariable in a term and returns a value of the original datatype:

```
inst :: Regular a ⇒ Subst a → SchemeOf a → a
inst subst = foldScheme findMetaVar to
  where
    findMetaVar x =
      maybe (error "inst: unbound metavariable")
            fst
            (lookup x subst)
```

Again, we lookup the corresponding term in the substitution. Since we construct a value of the original datatype, unbound metavariables are not allowed and result in a runtime error. A pattern functor value is converted to a value of the original datatype using *to*.

Finally, we combine the functions defined for matching a term and instantiating a term in the definition of *rewriteM*:

```
rewriteM :: (Regular a, Crush (PF a), Zip (PF a), Monad m)
         ⇒ Rule a → a → m a
rewriteM r term =
  do subst ← match (lhs r) term
     return (inst subst (rhs r))
```

We match the left-hand side of the rule to the given term, resulting in a substitution. Then, we use this substitution to instantiate the right-hand side of the rule. Since the matching process might fail, a monadic computation is returned. A similar library function is *rewrite*, defined in terms of *rewriteM*:

```
rewrite :: (Regular a, Crush (PF a), Zip (PF a))
        ⇒ Rule a → a → a
rewrite r term = maybe term id (rewriteM r term)
```

This function always succeeds since the original term is returned when rewriting fails.

Example

At this point, we can already use generic rewriting to implement the example presented in the introduction. However, as we will see, the rewriting library interface is not yet very user-friendly.

Recall that a rule for rewriting propositions has type Rule Prop. Hence, we define the rule *andContr* using a scheme representation:

```
andContr :: Rule Prop
andContr = p `pAnd` pNot p :⇝ pF
  where
    p = metaVar 1
```

We abbreviate the structure encodings of Prop values in order to improve readability. Functions *pF*, *pNot*, and *pAnd* are defined as follows:

```
pF :: SchemeOf Prop
pF = pf (Inr (Inr (Inl Unit)))
pAnd :: SchemeOf Prop → SchemeOf Prop → SchemeOf Prop
pAnd p q = pf (Inr (Inr (Inr (Inl (Id p :×: Id q)))))
pNot :: SchemeOf Prop → SchemeOf Prop
pNot p = pf (Inr (Inr (Inr (Inl (Id p)))))
```

Except for the application of *pf*, structure values resemble those used in the Regular instance for Prop. In fact, it is possible to define *pVar*, *pT*, and *pF* more concisely by using *toScheme*:

```
pVar :: String → SchemeOf Prop
pVar s = toScheme (Var s)
pT :: SchemeOf Prop
pT = toScheme T
pF :: SchemeOf Prop
pF = toScheme F
```

Unfortunately, *pNot*, *pAnd*, and *pOr* cannot be defined similarly. Consider the following failed attempt for *Not*:

```
badPNot :: SchemeOf Prop → SchemeOf Prop
badPNot p = toScheme (Not p)
```

This definition is not correct because the constructor *Not* cannot take a SchemeOf Prop value as an argument. It follows that we cannot use *Not* and *toScheme*, and we are forced to define *pNot* redundantly using the structure value of *Not*.

There are two problems with using generic rewriting as presented so far. First, in order to write concise rules, the user will need to define abbreviations for the structure representations of constructors, such as *pF* and *pAnd*. Second, such abbreviations force the user into relating constructors to their structure representations yet again, even though this is already made explicit in the Regular instance for the datatype.

2.4.3 Metavariables as Function Arguments

We have demonstrated how generic term rewriting is implemented in our library. A second contribution of this chapter is that we show how our library allows its users to specify rules using just the constructors of the original datatype definition. This is a clear improvement over the rewriting example above, because a user does not need to directly manipulate structure values. While such specifications are given using the original constructors, they still need to be transformed into SchemeOf values so that the rewriting machinery defined earlier can be used. We now discuss how to transform these rule specifications into the internal library representation of rules, or internal rules for short.

We start with the simpler case of rules that contain no metavariables. An example of such a rule is *notTrue*:

```
notTrue :: RuleSpec Prop
notTrue = Not T :~> F
```

Because the rule does not contain metavariables, it is sufficient to apply *toScheme* to perform the conversion:

```
rule0 :: Regular a => RuleSpec a -> Rule a
rule0 r = toScheme (lhs r) :~> toScheme (rhs r)
```

In our system, rules with metavariables are specified by functions from terms to rules. For example, the rule *andContr* is modified as follows to represent the metavariable:

```
andContr :: Prop -> RuleSpec Prop
andContr p = p :/\: Not p :~> F
```

A rule with one metavariable is represented by a function with one argument that returns a RuleSpec value, where the argument can be seen as a place-holder for the metavariable. This is convenient for programmers because metavariables are specified in the same way for different datatypes, rule specifications are more concise, scope checking is performed by the compiler, and implementation details such as variable names and the internal representation of terms remain hidden.

To use these specifications for rewriting, we have to transform them into the internal representation for rules. Consider the function that performs this transformation for rules with one metavariable:

```
rule1 :: (Regular a, Zip (PF a), LR (PF a)) => (a -> RuleSpec a) -> Rule a
rule1 r = introMetaVar (lhs . r) :~> introMetaVar (rhs . r)
```

Function *introMetaVar* performs the transformation from a function on terms to a term extended with metavariables. The value *r* is composed to yield either the left-hand side or right-hand side of the rule. The context LR (PF a) provides support for extending a term with metavariables, we defer the discussion of this predicate for the moment.

The function passed to *introMetaVar* is not allowed to inspect the metavariable argument: the metavariable can only be used as part of the constructed term. As our library relies on this property, the user should not inspect metavariables in rule definitions. For example, the following rule is invalid:

```
bogusRule :: Prop -> RuleSpec Prop
bogusRule (Var _) = T            :~> F
bogusRule p       = p :/\: Not p :~> F
```

Ideally, the restriction that metavariables are not inspected are encoded in the type system, so that rules like *bogusRule* are ruled out statically. However, to the best of

our knowledge, it is impossible to enforce this restriction without changing the user's datatype.

A rule function that uses but does not inspect its metavariable argument has the property that if two different values are passed to that function, the resulting rule values will only differ at the places where the metavariable argument occurs. The function *insertMetaVar* exploits this property and inserts a metavariable exactly at the places where the two term representations differ:

```
insertMetaVar :: (Regular a, Zip (PF a)) ⇒ MetaVar → a → a → SchemeOf a
insertMetaVar v x y =
  case fzip (insertMetaVar v) (from x) (from y) of
    Just str  → pf str
    Nothing → metaVar v
```

Function *insertMetaVar* traverses two pattern functor values in parallel using *fzip*, constructing a scheme (SchemeOf a) during the traversal. Whenever the two structures are different, a metavariable is returned. Otherwise, the recursive occurrences of the matching structures are traversed with *insertMetaVar v*.

We now define *introMetaVar*:

```
introMetaVar :: (Regular a, Zip (PF a), LR (PF a)) ⇒ (a → a) → SchemeOf a
introMetaVar f = insertMetaVar 1 (f left) (f right)
```

Function *insertMetaVar* requires that the two term arguments are different at metavariable occurrences, so we apply *f* to two values, *left* and *right*, such that they cause the failure of *fzip*. In particular, we want the following property to hold:

$$\forall f.\ fzip\ f\ (from\ left)\ (from\ right) \equiv Nothing$$

Informally, this property states that *left* and *right* produce values of the same type with different top-level constructors (or the same top-level constructor but different values at a non-recursive position). This follows from the fact that *fzip* only fails for incompatible pattern functor values, which represent the top-level constructors. It is important that *left* and *right* differ at the top-level (i.e., as soon as possible), otherwise metavariables would be inserted deeper than intended. Functions *left* and *right* are defined as follows:

```
left :: (Regular a, LR (PF a)) ⇒ a
left = to (leftf left)
right :: (Regular a, LR (PF a)) ⇒ a
right = to (rightf left)
```

These definitions use auxiliary functions *leftf* and *rightf* that generate a pattern functor value and convert it to the expected a using *to*. The arguments of *leftf* and *rightf* are the recursive occurrences, here we give the same argument in both cases (*left*) to emphasize

```
class LR f where
  leftf  :: a → f a
  rightf :: a → f a
instance LR Id where
  leftf  x = Id x
  rightf x = Id x
instance LRBase a ⇒ LR (K a) where
  leftf  _ = K leftb
  rightf _ = K rightb
instance LR Unit where
  leftf  _ = Unit
  rightf _ = Unit
instance (LR f, LR g) ⇒ LR (f :+: g) where
  leftf  x = Inl (leftf x)
  rightf x = Inr (leftf x)
instance (LR f, LR g) ⇒ LR (f :×: g) where
  leftf  x = leftf  x :×: leftf  x
  rightf x = rightf x :×: rightf x
```

Figure 2.7: *leftf* and *rightf* definitions

that the difference must be at top-level. These auxiliary functions are defined generically, that is, by induction on the structure of the pattern functor. Figure 2.7 shows the definitions of both functions as methods of the type class LR. The constant case uses an additional type class, which is shown in Figure 2.8.

The property for *fzip* above is restated for *leftf* and *rightf* as follows:

$$\forall f\ x\ y.\ fzip\ f\ (leftf\ x)\ (rightf\ y) \equiv Nothing$$

Note that the LR instances for Unit and Id do not satisfy the above property. We explain why we chose this suboptimal formulation. Ideally, we would prefer to enforce the above property statically for all pattern functors, and replace the LR instances for Unit and Id by weaker instances of the type class One, where One produces a single functor value. Then, we could write

```
instance (One f, One g) ⇒ LR (f :+: g) where ...
```

Unfortunately, the product case makes this scheme unrealistic:

```
instance (LR  f, One g) ⇒ LR (f :×: g) where ...
instance (One f, LR  g) ⇒ LR (f :×: g) where ...
```

To generate two different product values, at least one of the components should be inhabited by two different values. However, these two overlapping instances cannot

```
class LRBase a where
  leftb  :: a
  rightb :: a
instance LRBase Int where
  leftb  = 0
  rightb = 1
instance LRBase [a] where
  leftb  = []
  rightb = [error "Should never be inspected"]
```

Figure 2.8: *leftb* and *rightb* definitions

be expressed in Haskell. So, we adopt the current simpler approach, where, to satisfy the property above, we require the pattern functor to contain at least one sum or one constant case. Furthermore, note that the current instance of sum cannot be given as, for example, *Inl* $\perp$ and *Inr* $\perp$, because the function *to* may evaluate to $\perp$ when presented with such values. We call *leftf* recursively in both sum methods, to emphasize that different sum choices are enough for *zip* to fail.

With a slight generalization we allow the specification of rules with two metavariables. For example, consider the following rule:

```
deMorgan :: Prop → Prop → RuleSpec Prop
deMorgan p q = Not (p :∧: q) :⇝ Not p :∨: Not q
```

To use the *deMorgan* rule with our rewriting machinery we need a variant of $rule_1$ that handles two metavariables:

```
rule2 :: (Regular a, Zip (PF a), LR (PF a))
      ⇒ (a → a → RuleSpec a) → Rule a
rule2 r =
  introMetaVar2 (λx y → lhs (r x y)) :⇝
  introMetaVar2 (λx y → rhs (r x y))
```

The real work is carried out by $introMetaVar_2$.

```
introMetaVar2 :: (Regular a, Zip (PF a), LR (PF a))
              ⇒ (a → a → a) → SchemeOf a
introMetaVar2 f = term1 `mergeSchemes` term2
  where
    term1 = insertMetaVar 1 (f left left) (f right left  )
    term2 = insertMetaVar 2 (f left left) (f left  right)
```

As before, metavariables are inserted by *insertMetaVar*, but there is a slight complication: the rule specification now has two arguments rather than one. Because

insertMetaVar inserts only one variable at a time, we need to handle the two metavariables separately. Suppose that we handle the first metavariable, then we need two terms to pass to *insertMetaVar*. We apply f to different arguments for the first metavariable, and we supply the same argument for the second metavariable. In this way, differences in the generated terms arise only for the first metavariable. The second metavariable is handled in a similar way.

At this point we have two terms, each containing either one of the metavariables. We use the function *mergeSchemes* to combine the two schemes into one that contains both metavariables.

```
mergeSchemes :: Zip f ⇒ Scheme f → Scheme f → Scheme f
mergeSchemes p@(In x) q@(In y) =
  case (schemeView p, schemeView q) of
    (MetaVar i, _) → p
    (_, MetaVar j) → q
    _              → In (fzip′ mergeSchemes x y)
```

The generic function $fzip'$ is used to traverse both terms in parallel until a metavariable is encountered at either side. Then, the term at the other side can be discarded, for we are sure that it is just a *left*-value that was kept constant for the sake of handling one metavariable at a time.

2.4.4 Rules with an Arbitrary Number of Metavariables

Our technique for introducing metavariables in rules exhibits a clear pattern. It would be easy to define functions $rule_3$, $rule_4$, and so on to handle the cases for other fixed numbers of metavariables.

Alternatively, we can try and capture the general pattern in a class declaration and a suitable set of instance declarations. To this end, let us first examine this particular pattern more closely. In general, a rule involving n metavariables, for some natural number n, can be built by a function f that expects n arguments and produces a rule specification. To do so, we invoke the function n times, each time putting the value *right* into a specific argument position while keeping the value *left* in the remaining positions. Each of the n rule specifications obtained through this diagonal pattern are combined, by means of the function *insertMetaVar*, with a base value obtained by passing *left* values exclusively to f:

$$\underbrace{(f\ left\ left \cdots left)}_{base} \bowtie \underbrace{\begin{Bmatrix} (\ f\ right & left & \cdots & left\ ) \\ (\ f & left & right & \cdots & left\ ) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (\ f & left & left & \cdots & right\ ) \end{Bmatrix}}_{diag}$$

In this way, we obtain n rules that are, by repeated merging, combined into the single rule originally expressed by f.

First, we define a type class Builder that contains the family of functions that is used to build rules:

```
class Regular (Target a) ⇒ Builder a where
  type Target a :: ∗
  base      :: a → RuleSpec (Target a)
  diag      :: a → [RuleSpec (Target a)]
```

Apart from the methods *base* and *diag* that provide values obtained from invocations of each Builder-function with *left* and *right* arguments, the class contains an associated type synonym Target that holds the type targeted by the rule under construction. Some typical instances of the resulting type family are:

```
type Target (                    RuleSpec Prop) = Prop
type Target (          Prop → RuleSpec Prop) = Prop
type Target (Prop → Prop → RuleSpec Prop) = Prop
```

It remains to inductively define instances for building rewrite rules with an arbitrary number of metavariables. The base case, for constructing rules that do not involve any metavariables, is straightforward: such rules are easily built from functions that take no arguments and immediately return a RuleSpec:

```
instance Regular a ⇒ Builder (RuleSpec a) where
  type Target (RuleSpec a) = a
  base x                   = x
  diag x                   = [x]
```

Given an instance for the case involving n metavariables, the case for $(n+1)$ metavariables is, perhaps surprisingly, not much harder:

```
instance (Builder a, Regular b, LR (PF b)) ⇒ Builder (b → a) where
  type Target (b → a) = Target a
  base f              = base (f left)
  diag f              = base (f right) : diag (f left)
```

We now define a single function *rule* for constructing rewrite rules from functions over metavariable place-holders:

```
rule :: (Builder r, Zip (PF (Target r))) ⇒ r → Rule (Target r)
rule f = foldr1 mergeRules rules
  where
    mergeRules x y = mergeSchemes (lhs x) (lhs y) :⇝
                     mergeSchemes (rhs x) (rhs y)
    rules          = zipWith (ins (base f)) (diag f) [1..]
    ins x y v      = insertMetaVar v (lhs x) (lhs y) :⇝
                     insertMetaVar v (rhs x) (rhs y)
```

For instance, rules are now defined as follows:

```
notTrueRule, andContrRule, deMorganRule :: Rule Prop
notTrueRule    = rule (          Not T          :⇝ F)
andContrRule   = rule (λp    → p :∧: Not p    :⇝ F)
deMorganRule = rule (λp q → Not (p :∧: q) :⇝ Not p :∨: Not q)
```

To directly apply a rule constructed by a Builder, we compose *rule* and *rewrite*:

```
applyRule :: (Builder r, Rewrite (Target r)) ⇒ r → Target r → Target r
applyRule = rewrite ∘ rule
```

In summary, the Builder class and its instances provide a uniform approach to constructing rewrite rules that involve an arbitrary number of metavariables. Note that rules with different numbers of metavariables are, once built, uniformly typed and can, for example, be straightforwardly grouped together in a list or in any other data structure. Building rules by repeatedly comparing the results of different invocations of functions comes with some overhead, but for the typical case where the rule definition is in a constant applicative form, this amounts to a one-time investment.

2.4.5 Polishing the Interface

We now have a library for generic rewriting. However, the types of high-level functions contain too much implementation detail. For instance, recall the type of *rewriteM*:

```
rewriteM :: (Regular a, Crush (PF a), Zip (PF a), Monad m)
         ⇒ Rule a → a → m a
```

Function *rewriteM* exposes implementation details such as calls to generic *crush* and generic *zip* in its type signature. We hide these by defining a type class synonym Rewrite as follows:

```
class (Regular a, Crush (PF a), Zip (PF a), LR (PF a))
   ⇒ Rewrite a
```

For instance, the type of propositions is made an instance of Rewrite by declaring

```
instance Rewrite Prop
```

Using Rewrite, the type signature of *rewriteM* becomes:

```
rewriteM :: (Rewrite a, Monad m) ⇒ Rule a → a → m a
```

Some of the constraints of Rewrite are superfluous to some generic functions in the library. For example, the function *rule* only requires an instance of Zip. It might appear that the additional constraints on *rule* restrict its use unnecessarily, but this is not a problem in practice because functors built from sums and products already satisfy those constraints.

2.5 Performance

We have measured execution times of our generic rewriting library, mainly to measure how well the generic definitions perform compared to hand-written code for a specific datatype. We have tested our library with the proposition datatype from the introduction, extended with constructors for implications and equivalences. We have defined 15 rewrite rules, and we used these rules to bring the proposition to disjunctive normal form (DNF). This rewrite system is a realistic application of our rewriting library, and is very similar to the system that is used in an exercise assistant for e-learning systems (Heeren et al. 2008).

The library has been tested with four different strategies: such a strategy controls which rewrite rule is tried, and where. The strategies range from naive (i.e, apply some rule somewhere), to more involved strategy specifications that stage the rewriting and use all kinds of traversal combinators. QuickCheck (Claessen and Hughes 2000) is used to generate a sequence of random propositions, where the height of the propositions is bounded by five, and the same sequence is used for all test runs. Because the strategy highly influences how many rules are tried, we vary the number of propositions that has to be brought to disjunctive normal form depending on the strategy that is used. The following table shows for each strategy the number of propositions that are normalized, how many rules are successfully applied, and the total number of rules that have been fired:

strategy	terms	rules applied	rules tried	ratio
dnf-1	10,000	217,076	113,728,320	0.19%
dnf-2	50,000	492,114	22,716,336	2.17%
dnf-3	50,000	487,490	22,955,220	2.12%
dnf-4	100,000	872,494	19,200,407	4.54%

The final column shows the percentage of rules that succeeded: the numbers reflect that the simpler strategies fire more rules.

We compare the execution times of four different implementations for the collection of rewrite rules.

- **Pattern Matching (PM).** The first implementation defines the 15 rewrite rules as functions that use pattern matching. Obviously, this implementation suffers from the drawbacks that were mentioned in Section 2.2.1, making this version less suitable for an actual application. However, this implementation of the rules is worthwhile to study because Haskell has excellent support for pattern matching, which will likely result in efficient code.

- **Specialized Rewriting (SR).** We have also written a specialized rewriting function that operates on propositions. Because the Prop datatype does not have a constructor for metavariables, we have reused the *Var* constructor for this purpose, thus mixing object variables with metavariables.

- **Pattern-functor View (PV).** Here we implemented the rules using the generic functions for rewriting that were introduced in this chapter. The instance for

the Regular type class is similar to the declaration in Figure 2.3, except that a balanced encoding was used for the constructors of the Prop datatype to make it more efficient.

- **Fixed-point View (FV).** The last version also uses the generic rewriting approach, but it uses a different structural representation called the fixed-point view (Holdermans et al. 2006). Like the pattern-functor view, this view makes recursion explicit in a datatype, but additionally, every recursive occurrence is mapped to its structural counterpart. In this view, *from* and *to* would have the following types:

  $$\begin{aligned} &\mathit{from} :: \mathsf{a} \to \mathsf{Fix}\,(\mathsf{PF}\;\mathsf{a}) \\ &\mathit{to} \quad :: \mathsf{Fix}\,(\mathsf{PF}\;\mathsf{a}) \to \mathsf{a} \end{aligned}$$

  We include this view in our measurements because such deep structure mapping is common in generic programs that deal with regular datatypes (Jansson and Jeuring 1997). Indeed, the fixed-point view was used in our library before we switched to the current structure representation.

All test runs were executed on an Apple MacBook Pro with a 2.2 GHz Intel Core 2 Duo processor, 2 Gb SDRAM memory, and running MacOS X 10.5.3. The programs were compiled with GHC version 6.8.3 with all optimizations enabled (using the -O2 compiler flag). Execution times were measured using the `time` shell command, and were averaged over three runs. The following table shows the execution time in seconds for each implementation of the strategies:

strategy	PM	SR	PV	FV
dnf-1	6.31	16.75	56.78	70.69
dnf-2	3.49	6.37	23.66	30.24
dnf-3	3.52	6.42	23.82	30.26
dnf-4	5.72	9.14	27.82	37.65

Because the strategies normalize a varying number of terms, it is hard to draw any conclusion from results of different rows. The table shows that the pattern-matching approach (PM) is significantly faster compared to the other approaches. The specialized rewriting approach (SR) adds observability of the rewrite rules, at the cost of approximately doubling the execution time. The two generic versions, when compared to the SR approach, suffers from a slowdown of a factor 3 to 4. This slowdown is probably due to the conversions from and to the structure representation of propositions. The approach with a fixed-point view (FV) is less efficient than using the pattern-functor view (PV) since the latter only converts a term when this is really required. This is reflected in the recursive embedding-projection pair for the FV in contrast to the non-recursive embedding-projection pair of the PV.

Execution time is consumed not only by the rewrite rules, but also by the strategy-controlled traversals over the propositions. These traversal functions, such as *transform*

from the introduction, can of course be defined generically. The execution times presented before use specialized traversal functions for the Prop datatype, but we have repeated the experiment using the generic traversal definitions. Although traversal combinators are not the focus of this chapter, it is interesting to measure how much impact this change has on performance. The following table shows the execution times for the different implementations that now make use of generic traversal combinators:

strategy	PM	SR	PV	FV
dnf-1	10.56	20.50	61.95	91.12
dnf-2	11.18	14.40	32.21	54.86
dnf-3	11.38	14.51	32.21	55.44
dnf-4	9.04	12.53	31.56	46.08

First, the observations made for the table presented earlier still hold. By comparing the two tables point-wise it can be concluded that the generic traversal functions are slower. The relative increase in execution time for the versions that already used generic definitions (that is, PV and FV) is, however, lower compared to the non-generic versions (PM and SR). Furthermore, the additional cost of making the traversal generic depends on the strategy being used, since the strategies use different combinators.

So what can be concluded from these tests? The tests do not offer spectacular new results, but rather confirm that observability of rules comes at the expense of loss in runtime efficiency. Furthermore, generic definitions introduce some additional overhead. The trade-off between efficiency and genericity depends on the application at hand. For instance, the library would be suitable for the online exercise assistant, because runtime performance is less important in such a context. We believe that improving the efficiency of generic library code is an interesting area for future research. By inlining and specializing generic definitions, and by applying partial evaluation techniques, we expect to get code that is more competitive to the hand-written definitions for a specific datatype. Fusion techniques (Alimarine and Smetsers 2005; Coutts et al. 2007) can help to eliminate some of the conversions between a value and its structure counterpart.

2.6 Related Work

The generic rewriting library introduced in this paper can be viewed as a successor to the library given by Jansson and Jeuring (2000); we discussed the relation with this library in the introduction.

Libraries that provide generic traversal combinators, such as Strafunski (Lämmel and Visser 2002), Scrap Your Boilerplate (Lämmel and Peyton Jones 2003), Uniplate (Mitchell and Runciman 2007b), a pattern for almost compositional functions (Bringert and Ranta 2006), and probably more, can be used to define rewrite rules in the form of functions. This has the disadvantages we describe in Section 2.2, the most important of which is that it is impossible to document, test, and analyse rewrite rules. The advantage of using functions instead of rules is that datatypes do not have to be extended with metavariables: we reuse program-level variables for metavariables in our rules.

The *match* function is a simple variant of generic unification functions (Jansson and Jeuring 1998a; Sheard 2001). Our library cannot be easily generalized to perform unification and stay user-friendly, because the result of unification may contain values of datatypes extended with metavariables, and hence a user would have to deal with structure values. On the other hand, we could use two-level types as in the work from Sheard but require less work from the user, because most functionality can be obtained generically from the structure representation of the two-level types. Furthermore, we could easily adapt our library to use mutable variables, as in Sheard's work, to improve performance.

Brown and Sampson (2008) implement generic rewriting using the Scrap Your Boilerplate library. Rule schemes are described in a special purpose datatype that does not depend on the type of values being rewritten. In contrast to our system, rules are not typed and hence invalid rules are only detected at runtime. This system can handle rewriting of a system of datatypes while our library is limited to a single regular datatype. However, we know how to lift this restriction in a type-safe way, see Section 2.7.

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

2.7 Conclusions and Further Work

We have shown the interface and implementation of a generic library for rewriting. Our library overcomes problems in previous generic rewriting libraries: users do not have to adapt the datatype on which they want to apply rewriting, they do not need knowledge of how the generic rewriting library has been implemented, and they can document, test, and analyse their rules. The performance of our library is not as good as that of hand-written datatype-specific rewriting functions, but we think the loss of performance is acceptable for many applications.

One of the most important limitations of the library described in this chapter is that it only works for datatypes that can be represented by means of a fixed-point. Such datatypes are also known as regular datatypes. This is a severe limitation, which implies that we cannot apply the rewriting library to nested datatypes or systems of (mutually recursive) datatypes. Indeed, many real-world applications involve such systems: examples include systems of linear equations and the abstract syntax of expressions that may contain declarations that, in turn, may consist of expressions again. However, with some additional machinery we can overcome this limitation and so we have an implementation of the rewriting library that enables generic rewriting on systems of datatypes. The techniques used are rather sophisticated and apply to several other problems as well, such as the Zipper. A thorough explanation of the underlying ideas and implementation can be found in Chapter 3.

Our library requires that rules do not inspect their metavariable argument. For in-

stance, we do not allow arbitrary function applications in the right-hand side of a rule, contrary to rules that are defined as functions with pattern matching. Another limitation of our library is that rules cannot have preconditions, for example, that a proposition matched against a metavariable is in disjunctive normal form. Extending rewrite rules with preconditions remains future work.

Currently, we are also working on generating test data generically. The left-hand side of a rewrite rule can be used as a template for test data generation to improve the testing coverage. We plan to use it to provide a testing framework for users of our library. This functionality can be easily added to our library: all it takes to define a new generic function is declaring a type class and providing a set of inductively defined instances.

Acknowledgments

This work was made possible by the support of the SURF Foundation, the higher education and research partnership organization for Information and Communications Technology (ICT). Please visit `http://www.surf.nl` for more information about SURF.

This work has been partially funded by the Technology Foundation STW through its project on "Demand Driven Workflow Systems" (07729), and by the Netherlands Organisation for Scientific Research (NWO), through its projects on "Real-life Datatype-Generic Programming" (612.063.613) and "Scriptable Compilers" (612.063.406).

The authors would like to thank Chris Eidhof and Sebastiaan Visser for their work on testing rewrite rules by means of generic generation of test data, and Andres Löh for productive discussions on this work. Finally, the authors are indebted to the anonymous reviewers for their useful suggestions.

3 Fixed Points for Mutually Recursive Datatypes

Many datatype-generic functions need access to the recursive positions in the structure of the datatype, and therefore adopt a fixed point view on datatypes. Examples include variants of fold that traverse the data following the recursive structure, or the zipper data structure that enables navigation along the recursive positions. However, Hindley-Milner-inspired type systems with algebraic datatypes make it difficult to express fixed points for anything but regular datatypes. Many real-life examples such as abstract syntax trees are in fact systems of mutually recursive datatypes and therefore excluded.

In this chapter, we introduce a technique that allows a fixed-point view for systems of mutually recursive datatypes. We present our technique using Haskell and recent language extensions such as generalized algebraic datatypes (GADTs) and type families. We demonstrate that our approach is widely applicable by giving several examples of generic functions for this view, most prominently the Zipper.

3.1 Introduction

One of the most important activities in software development is *structuring data*. Many programming methods and software development tools center around creating a datatype (or XML schema, UML model, class, grammar, etc.). Once the structure of the data has been designed, a software developer adds *functionality* to the datatypes. There is always some functionality that is specific for a datatype, and part of the reason why the datatype has been designed in the first place. Other functionality is similar or even the same on many datatypes. Examples of such functionality are:

- in a large datatype, looking for occurrences of a particular constructor (e.g., for representing variables) for which we want to do something, ignoring the rest of the value;
- functions that depend only on the *structure* of the datatype, such as the equality function;
- adapting the code after datatypes have changed or evolved.

Generic programming addresses these high-level programming patterns. We also use the term datatype-generic programming to distinguish the field from Java generics, Ada generic packages, generic programming in C++ STL, etc. Using generic programming,

we can easily implement traversals in which a user is only interested in a small part of a possibly large value, functions which are naturally defined by induction on the structure of datatypes, and functions that automatically adapt to a changing datatype.

Generic programming grew out of category theory, in which datatypes are represented as initial algebras or fixed points of functors (Hagino 1987; Malcolm 1990; Meijer et al. 1991). A functor is a datatype of kind $* \to *$ together with a *map* function. Fixed points are represented by an instance of the Fix datatype:

$$\textbf{data}\ \mathsf{Fix}\ \mathsf{f} = \mathit{In}\ (\mathsf{f}\ (\mathsf{Fix}\ \mathsf{f}))$$

and functors can be constructed using a limited set of datatypes for sums and products.

The set of datatypes that can be represented by means of Fix is limited, and does not include mutually recursive datatypes and nested datatypes (Bird and Meertens 1998). Early generic programming systems based on a fixed-point view, such as PolyP (Jansson and Jeuring 1997), inherit these restrictions. The lack of support for mutually recursive datatypes is particularly limiting, because most larger systems are described by several datatypes with complex dependencies.

Partially to overcome such limitations, several other approaches to generic programming have been developed, among them Generic Haskell (Hinze 2000c; Löh 2004; Löh et al. 2008) and Scrap Your Boilerplate (Lämmel and Peyton Jones 2003). These approaches do not use fixed points to represent datatypes, and can handle mutually recursive types, and many or all nested types. However, they have no knowledge about the recursive structure of datatypes.

Quite a number of generic functions need information about the recursive structure of datatypes. Examples of such functions are the recursion schemes such as *fold* and its variants (Meijer et al. 1991), upwards and downwards accumulations (Bird et al. 1996; Gibbons 1998), unification (Jansson and Jeuring 1998a), rewriting and matching functions (Jansson and Jeuring 2000), functions for selecting subexpressions (Steenbergen et al. 2008), pattern matching (Jeuring 1995), design patterns (Gibbons 2006), the Zipper and its variants (Huet 1997; McBride 2001; Hinze et al. 2004; Morris et al. 2006; McBride 2008), etc. The recursive structure of datatypes also plays an important role when transforming programs or proving properties about programs, such as the fold fusion theorem (Malcolm 1990), the generic approximation lemma (Hutton and Gibbons 2001), and the acid-rain theorem (Takano and Meijer 1995).

In a generic programming system that does not use fixed points, such functions cannot be defined properly. In Generic Haskell, views on datatypes have been added, including a fixed-point view on datatypes (Holdermans et al. 2006): a programmer can choose to write a function that does not need access to the recursive positions and works on many datatypes, or to write a function that requires the fixed-point view, but that function is again restricted to a rather small set of datatypes.

The problem of representing a system of mutually recursive datatypes by means of an initial algebra or a fixed point has been studied before (Malcolm 1990). However, such solutions cannot be used as a base for generic programming on current Hindley-Milner-inspired type systems. The number of functors in the fixed point and their arity

is the same as the number of mutually recursive datatypes represented. In a language like Haskell, this pattern can only be captured by introducing new datatypes for every arity. As a consequence, we would have to duplicate the machinery over and over again, defeating the very purpose of generic programming.

This chapter makes the following contributions:

- We show how to represent a system of (arbitrarily many) mutually recursive datatypes using a fixed-point view in Haskell, using extensions to the Haskell language, most prominently generalized algebraic datatypes (Peyton Jones et al. 2006), type families (Chakravarty et al. 2005) and rank-2 types. Despite these extensions, the technique is easy to use for the programmer, and does not require more effort than other approaches to generic programming (Section 3.3).
- We show that our technique is widely applicable by presenting several generic algorithms that work on systems of mutually recursive datatypes. In Section 3.4, we demonstrate how to define recursion schemes such as *compos* (Bringert and Ranta 2006) and *fold*. In Section 3.5, we define a generic Zipper data structure (Huet 1997). In Section 3.6, we explain how to perform generic matching, which constitutes an important prerequisite for generic rewriting (Jansson and Jeuring 2000; Van Noort et al. 2008).

Our contributions are not necessarily limited to programming languages such as Haskell. We think that, in some applications, our technique may be simpler to use than those already solving the same problem in more expressive programming languages. To our knowledge, we are the first to show how to conveniently implement generic functions that need access to the recursive structure of datatypes on mutually recursive datatypes. For example, compiler writers get immediate access to generic matching, rewriting, and folding functions for their abstract syntax. The convenience is due to the lightweight nature of our approach: no generators or full dependent types are needed. Furthermore, our approach is non-invasive: the definitions of large systems of datatypes need not be modified in order to use generic programming.

In the remaining sections of this chapter, we discuss related work (Section 3.7) and conclusions (Section 3.8).

3.2 Fixed points for representing regular datatypes

Before we present fixed points for systems of mutually recursive datatypes, we review fixed points for regular datatypes. A functor is a datatype of kind $* \to *$ for which we can define a *map* function. Fixed points are represented by an instance of the Fix datatype:

$$\textbf{data}\ \mathsf{Fix}\ \mathsf{f} = \mathit{In}\ \{\mathit{out} :: (\mathsf{f}\ (\mathsf{Fix}\ \mathsf{f}))\}$$

Haskell's record notation is used to introduce the selector function $\mathit{out} :: \mathsf{Fix}\ \mathsf{f} \to \mathsf{f}\ (\mathsf{Fix}\ \mathsf{f})$. Using Fix, we can represent the following datatype for simple arithmetic expressions

```
data Expr = Const Int | Add Expr Expr | Mul Expr Expr
```

by its *pattern functor*:

```
data PF_Expr r = ConstF Int | AddF r r | MulF r r
type Expr' = Fix PF_Expr
```

Given a *map* function for $\mathsf{PF}_{\mathsf{Expr}}$, many recursion schemes (for example, *fold* and *unfold*) can be easily defined on Expr'.

3.2.1 Building functors systematically

It is not necessary to define a specific *map* function for each and every pattern functor, though – as long as we build functors using a fixed set of datatypes:

```
data K a       r = K a
data I         r = I r
data (f :×: g) r = f r :×: g r
data (f :+: g) r = L (f r) | R (g r)
infixr 7 :×:
infixr 6 :+:
```

We define a generic *map* function by declaring the following instances of the type class Functor:

```
class Functor f where
  fmap :: (a → b) → f a → f b
instance Functor I where
  fmap f (I r) = I (f r)
instance Functor (K a) where
  fmap _ (K x) = K x
instance (Functor f, Functor g) ⇒ Functor (f :+: g) where
  fmap f (L x) = L (fmap f x)
  fmap f (R y) = R (fmap f y)
instance (Functor f, Functor g) ⇒ Functor (f :×: g) where
  fmap f (x :×: y) = fmap f x :×: fmap f y
```

Now, if expressions are represented by the functor $\mathsf{PF}_{\mathsf{Expr}}$

```
type PF_Expr = K Int :+: (I :×: I) :+: (I :×: I)
type Expr'   = Fix PF_Expr
```

we get *fmap* for expressions for free.

Datatypes, such as Expr, whose recursive structure can be represented by a polynomial functor (consisting of sums, products and constants) are often called regular

datatypes. This uniform encoding allows us to define functions that work on all regular datatypes:

$$
\begin{aligned}
&\mathit{fold} :: \mathsf{Functor\ f} \Rightarrow (\mathsf{f\ a} \to \mathsf{a}) \to \mathsf{Fix\ f} \to \mathsf{a} \\
&\mathit{fold}\ f = f \circ \mathit{fmap}\ (\mathit{fold}\ f) \circ \mathit{out} \\
&\mathit{unfold} :: \mathsf{Functor\ f} \Rightarrow (\mathsf{a} \to \mathsf{f\ a}) \to \mathsf{a} \to \mathsf{Fix\ f} \\
&\mathit{unfold}\ f = \mathit{In} \circ \mathit{fmap}\ (\mathit{unfold}\ f) \circ f
\end{aligned}
$$

3.2.2 Representations of regular datatypes

Unfortunately, functions *fold* and *unfold* can only be used on regular datatypes that are already encoded as fixed points of functors. In other words, they work on values of type Expr′, but not on values of type Expr. In practice, being forced to work with Expr′ rather than Expr is inconvenient: it is much easier to work with the user-defined, appropriately named constructors of Expr than with the structural constructors of Expr′. Therefore, some generic programming languages automatically generate mappings that relate datatypes such as Expr with their structure representation counterparts (Expr′) (Jansson and Jeuring 1997; Löh 2004; Holdermans et al. 2006).

In Haskell we can express such mappings by means of a type class. We define the type class Regular to encode the recursive structure of regular datatypes:

```
type family PF a :: * -> *
class Functor (PF a) => Regular a where
  from :: a -> (PF a) a
  to   :: (PF a) a -> a
```

The functor type of a regular datatype is given by the *pattern functor* PF, a *type family*: for different instantiations of the type index a, we can provide different definitions of PF a. The pattern functor PF corresponds to PolyP's type constructor FunctorOf (Jansson and Jeuring 1997; Norell and Jansson 2004). The two methods *from* and *to* embed regular datatypes into their pattern functor-based representation.

Here is the Regular instance for expressions:

```
type instance PF Expr = PF_Expr
instance Regular Expr where
  from (Const i)   = L     (K i)
  from (Add e e')  = R (L (I e :×: I e'))
  from (Mul e e')  = R (R (I e :×: I e'))
  to (L     (K i))         = Const i
  to (R (L (I e :×: I e'))) = Add e e'
  to (R (R (I e :×: I e'))) = Mul e e'
```

Note that *from* and *to* transform only the top layer of a value. If desired, we can convert between Expr and Expr′ by recursively applying *from* and *to*.

We can now rewrite *fold* and *unfold* such that they work on instances of Regular (thus in particular, on Expr rather than Expr′):

```
fold :: Regular a ⇒ (PF a b → b) → a → b
fold f = f ∘ fmap (fold f) ∘ from
unfold :: Regular a ⇒ (b → PF a b) → b → a
unfold f = to ∘ fmap (unfold f) ∘ f
```

Another recursion scheme we can define is *compos* (Bringert and Ranta 2006). Much like *fold*, it traverses a data structure and performs operations on the children. There are different variants of *compos*, the simplest is equivalent to PolyP's *mapChildren* (Jansson and Jeuring 1998a): it applies a function of type a → a to all children. This parameter is also responsible for performing the recursive call, because *compos* itself is not recursive:

```
compos :: Regular a ⇒ (a → a) → a → a
compos f = to ∘ fmap f ∘ from
```

3.3 Fixed points for mutually recursive datatypes

This section investigates representing systems of mutually recursive datatypes as fixed points. We first show why fixed points for single datatypes do not easily generalize to the situation of multiple datatypes. Subsequently, we present a solution to that problem, culminating in a library for representing systems of datatypes.

3.3.1 The problem

Consider the following extended version of our Expr datatype:

```
data Expr = Const Int
          | Add   Expr Expr
          | Mul   Expr Expr
          | EVar  Var
          | Let   Decl Expr
data Decl = Var := Expr
          | Seq Decl Decl
type Var  = String
```

We cannot represent Expr as before: the pattern functor of Expr exposes the direct recursive occurrences of Expr, but every occurrence of Decl is treated as a constant, even though declarations can contain expressions again.

One possibility, presented by Swierstra et al. (1999), is to use a different fixed point datatype, which abstracts over bifunctors of kind $* \to * \to *$ rather than functors of kind $* \to *$:

```
data Fix2 f g = In2 (f (Fix2 f g) (Fix2 g f))
```

Representing a set of three mutually recursive datatypes in this style requires a fixed point that takes three arguments, and so on. As a result, we cannot use one set of datatypes to build functors and define generic functions just for that one set. Instead, we basically have to duplicate the same machinery over and over again, defeating the very purpose of generic programming. Furthermore, systems of datatypes can be very large, so we cannot hope that using a limited amount of Fix-variants will suffice in practice.

3.3.2 Intermediate step: using a GADT

We can encode a system of (arbitrarily many) mutually recursive datatypes using a GADT (Bringert and Ranta 2006). For instance, we can replace the definitions in Section 3.3.1 by the GADT defined as the union of the previously separate datatypes:

```
data AST :: * → * where
  Const :: Int → AST Expr
  Add   :: AST Expr → AST Expr → AST Expr
  Mul   :: AST Expr → AST Expr → AST Expr
  EVar  :: AST Var → AST Expr
  Let   :: AST Decl → AST Expr → AST Expr
  (:=)  :: AST Var → AST Expr → AST Decl
  Seq   :: AST Decl → AST Decl → AST Decl
  V     :: String → AST Var
```

where Expr, Decl and Var are dummy types used to distinguish the categories, and, in the following, these do no longer refer to the original definitions:

```
data Expr
data Decl
data Var
```

There is a choice in the definition of AST. We can include Var in our set of mutually recursive datatypes, as we do here. Alternatively, we can just replace all occurrences of AST Var above with String, save the definition of the Var dummy type and the *V* constructor. However, we then cannot operate on variables directly in our generic functions.

It turns out that it is easier to think about the fixed point view in terms of the above GADT. In other words, we have reduced the problem of finding a fixed point for mutually recursive datatypes to that of finding a fixed point for a GADT. In the following, we "manually" define a pattern functor for the GADT, and we next show how to represent the structure of this pattern functor. Finally (in Section 3.3.5), we show how to use the same idea with the original system of datatypes.

3.3.3 Manually derived GADT pattern functor

We can define the pattern functor of our GADT directly, without using sums and products. The idea is to repeat the definition of AST given above, but now abstracting over the recursive positions.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{PF_{AST}}\ (\mathsf{r} :: * \to *) :: * \to * \ \textbf{where}\\
&\quad \mathit{ConstF} :: \mathsf{Int} \to \mathsf{PF_{AST}}\ \mathsf{r}\ \mathsf{Expr}\\
&\quad \mathit{AddF} \ :: \mathsf{r}\ \mathsf{Expr} \to \mathsf{r}\ \mathsf{Expr} \to \mathsf{PF_{AST}}\ \mathsf{r}\ \mathsf{Expr}\\
&\quad \mathit{MulF} \ :: \mathsf{r}\ \mathsf{Expr} \to \mathsf{r}\ \mathsf{Expr} \to \mathsf{PF_{AST}}\ \mathsf{r}\ \mathsf{Expr}\\
&\quad \mathit{EVarF} \ :: \mathsf{r}\ \mathsf{Var} \to \mathsf{PF_{AST}}\ \mathsf{r}\ \mathsf{Expr}\\
&\quad \mathit{LetF} \ :: \mathsf{r}\ \mathsf{Decl} \to \mathsf{r}\ \mathsf{Expr} \to \mathsf{PF_{AST}}\ \mathsf{r}\ \mathsf{Expr}\\
&\quad \mathit{BindF} \ :: \mathsf{r}\ \mathsf{Var} \to \mathsf{r}\ \mathsf{Expr} \to \mathsf{PF_{AST}}\ \mathsf{r}\ \mathsf{Decl}\\
&\quad \mathit{SeqF} \ :: \mathsf{r}\ \mathsf{Decl} \to \mathsf{r}\ \mathsf{Decl} \to \mathsf{PF_{AST}}\ \mathsf{r}\ \mathsf{Decl}\\
&\quad \mathit{VF} \ :: \mathsf{String} \to \mathsf{PF_{AST}}\ \mathsf{r}\ \mathsf{Var}
\end{aligned}
$$

A recursive occurrence is referred to by the variable r, which is indexed by the type of the subtree, and hence has kind $* \to *$. The fixed point datatype is changed to account for the addition of the type index in the functor – but note that the kind of HFix is independent of the number of mutually recursive datatypes:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{HFix}\ (\mathsf{f} :: (* \to *) \to * \to *)\ \mathsf{a} = \mathit{HIn}\ (\mathsf{f}\ (\mathsf{HFix}\ \mathsf{f})\ \mathsf{a})\\
&\textbf{type}\ \mathsf{AST}' = \mathsf{HFix}\ \mathsf{PF_{AST}}
\end{aligned}
$$

At the top level, the index argument a of HFix is passed to the functor f. At each recursive point, the index can be changed as desired: for example, *LetF* dictates that the index of the first recursive call is Decl, and the index of the second recursive call is Expr. Using the HFix-based encoding, the expression $\mathit{Let}\ (\mathit{V}\ \texttt{"x"} := \mathit{Const}\ 2)\ (\mathit{EVar}\ (\mathit{V}\ \texttt{"x"}))$ can be represented by

$$
\begin{aligned}
&\mathit{HIn}\ (\mathit{LetF}\ (\mathit{HIn}\ (\mathit{BindF}\ (\mathit{HIn}\ (\mathit{VF}\ \texttt{"x"}))\ (\mathit{HIn}\ (\mathit{ConstF}\ 2))))\\
&\qquad\qquad (\mathit{HIn}\ (\mathit{EVarF}\ (\mathit{HIn}\ (\mathit{VF}\ \texttt{"x"}))))))
\end{aligned}
$$

3.3.4 Representing the pattern functor

To support generic programming on mutually recursive datatypes in a way similar to that shown for regular ones in Section 3.2, we want to describe our datatypes by means of a pattern functor expressed in terms of a small number of building blocks, such as binary sums and products.

A failed attempt

It is not enough to just encode AST by ignoring the type index (and thus changing the kind of $\mathsf{PF_{AST}}$):

```
type PF_AST = K Int    :+:   -- Const
            (I :×: I) :+:   -- Add
            (I :×: I) :+:   -- Mul
            I         :+:   -- EVar
            (I :×: I) :+:   -- Let
            (I :×: I) :+:   -- :=
            (I :×: I) :+:   -- Seq
            K String        -- V
type AST'  = Fix PF_AST
```

This way, we have a problem with the conversion between the original datatype and the pattern functor: we can write *from* (it just forgets the index), but *to* is not typeable (it has to recover the index) – consider the case for *Const*:

$$to :: \mathsf{AST}' \to \mathsf{AST\ a}$$
$$to\ (L\ (K\ i)) = Const\ i$$

The application of *Const* produces an AST Expr, but the type of *to* requires the result to be polymorphic in the index. The pattern match does not refine the type of the case, as no GADTs are involved in the match. And hence this function is not typeable. The (unsatisfactory) solution would be to make the *to* function specific to each AST category:

$$toExpr :: \mathsf{AST}' \to \mathsf{AST\ Expr}$$
$$toExpr\ (In\ (L\ (K\ i))) = Const\ i$$

But this would make *toExpr* partial since it cannot handle representations of AST Decl. Worse, many static guarantees, such as performing *fmap* over an expression should return an expression, would no longer be enforced by the type system.

A faithful representation

In order to represent the GADT AST faithfully, we have to keep track of the index, as we did in Section 3.3.3. We therefore perform two generalizations.

First, every type constructor gets an additional argument ix, namely, the index type of the tree being represented. Second, we add an argument xi to I to indicate the index type on which we recurse. Since the recursion may be on a different index, xi need not be the same as ix.

```
data I xi       r ix = I (r xi)
data K a        r ix = K a
data (f :+: g) r ix = L (f r ix) | R (g r ix)
data (f :×: g) r ix = f r ix :×: g r ix
```

There is one missing piece in the representation. How is the choice between an Expr constructor and a Decl constructor represented? A sum can be used for the choice, but

it cannot constrain ix to either type. For this purpose, we define the following GADT (:▷:). For a type expression f :▷: xi, we say that the structure representation f is tagged with the tag xi.

```
infix 6 :▷:
data (f :▷: xi) r ix where
  Tag :: f r ix → (f :▷: ix) r ix
```

Tagging a structure representation constrains the index ix to be the same as the tag argument xi. We are now ready to give a structure representation for AST:

```
type PF_AST = K Int                 :▷: Expr :+:   -- Const
            (I Expr :×: I Expr)     :▷: Expr :+:   -- Add
            (I Expr :×: I Expr)     :▷: Expr :+:   -- Mul
            I Var                   :▷: Expr :+:   -- EVar
            (I Decl :×: I Expr)     :▷: Expr :+:   -- Let
            (I Var :×: I Expr)      :▷: Decl :+:   -- :=
            (I Decl :×: I Decl)     :▷: Decl :+:   -- Seq
            K String                :▷: Var        -- V
```

The differences with the representation of Expr in Section 3.2 are that I takes the index type of the recursive position, and each constructor representation is tagged with the AST category that it represents. The conversion functions hardly differ, except for dealing with tags:

```
from :: AST a → PF_AST AST a
from (Const i)  = L                    (Tag (K i))
from (Add e e') = R (L                 (Tag (I e :×: I e')))
from (Mul e e') = R (R (L              (Tag (I e :×: I e'))))
from (EVar x)   = R (R (R (L           (Tag (I x)))))
from (Let d e)  = R (R (R (R (L        (Tag (I d :×: I e))))))
from (x := e)   = R (R (R (R (R (L     (Tag (I x :×: I e)))))))
from (Seq d d') = R (R (R (R (R (L (Tag (I d :×: I d'))))))))
from (V x)      = R (R (R (R (R (R (Tag (K x)))))))
```

```
to :: PF_AST AST a → AST a
to (L                 (Tag (K i)))          = Const i
to (R (L              (Tag (I e :×: I e'))))= Add e e'
to (R (R (L           (Tag (I e :×: I e'))))) = Mul e e'
to (R (R (R (L        (Tag (I x)))))) = EVar x
to (R (R (R (R (L     (Tag (I d :×: I e))))))) = Let d e
to (R (R (R (R (R (L  (Tag (I x :×: I e))))))))= x := e
to (R (R (R (R (R (R (L (Tag (I d :×: I d'))))))))))= Seq d d'
to (R (R (R (R (R (R (Tag (K x)))))))))= V x
```

The first argument of $\mathsf{PF}_{\mathsf{AST}}$ specifies that the recursive occurrences are AST-values. Both *from* and *to* are now typeable. The pattern matches on *Tag* refine the type of the equations. For instance, in the case for *Const* the pattern match on *Tag* indicates that a must be equal to Expr on the right hand side. It follows that *Const i* is of type AST Expr.

3.3.5 Eliminating the GADT

Now, we turn our attention back to the AST definitions given in Section 3.3.1. Interestingly, we do not need the GADT AST. We use it as an inspiration to define $\mathsf{PF}_{\mathsf{AST}}$, but then map directly between $\mathsf{PF}_{\mathsf{AST}}$ and the original datatypes.

The library for representing systems

We can make a type class specific to our AST type to perform this mapping, but ultimately, we want a library that works with many systems of datatypes, therefore we declare:

```
class Ix s ix where
  from  :: ix → Str s ix
  to    :: Str s ix → ix
  index :: s ix
```

The parameter s indicates the system of datatypes we are working with, and the predicate Ix s ix expresses that ix is an index of system s. The structural representation of a type Str s ix is expressed in terms of a generalized pattern functor type family:

```
type family PF s :: (* → *)   -- datatype system s
                 → (* → *)    -- recursive occurrences r
                 → *          -- index type ix
                 → *
type Str s ix = (PF s) s I* ix
```

A datatype is represented by the pattern functor of system s constrained to represent values of type ix. Note that the r argument in Str is instantiated to the type I_*:

```
data I* a = I*{unI* :: a}
```

The type I_* behaves as the identity on types so that recursive occurrences inside the functor are stored "as is". Although the definition of I_* is essentially the same as that of I in Section 3.2, we give it a different name to highlight the different role that it plays in this representation.

The structure type constructors are extended once more, passing around the information about the system s being represented:

```
data I xi        (s :: * → *) (r :: * → *) ix  where
  I :: Ix s xi ⇒ r xi → I xi s r ix
data K a         (s :: * → *) (r :: * → *) ix = K a
data (f :+: g)   (s :: * → *) (r :: * → *) ix = L (f s r ix)
                                               | R (g s r ix)
data (f :×: g)   (s :: * → *) (r :: * → *) ix = f s r ix :×: g s r ix
data (f :▷: ix)  (s :: * → *) (r :: * → *) ix' where
  Tag :: f s r ix → (f :▷: ix) s r ix
```

Apart from the lifting, the only change in the type constructors is in the definition of I, where we require the type used in the recursion to be part of the datatype system, introducing the constraint Ix s xi.

It is possible to simplify this new representation by always assuming that r is I_*. As a result, we would not need the argument r and type I_* anymore, and the argument of *I* could just have type xi rather than the current r xi. The resulting representation would be simpler to use but also more limited. While it could be used for applications such as *compos* and the Zipper, it would be useless for applications that change the type of recursive occurrences such as *fold*, *unfold*, and generic rewriting. Therefore, we prefer to use the more general representation throughout this chapter.

Instantiating the library to a system

We now illustrate our approach by giving a representation for our datatype system and defining the conversion functions. First, we define the datatype that represents the datatypes in our system:

```
data AST :: * → * where
  Expr :: AST Expr
  Decl :: AST Decl
  Var  :: AST Var
```

The datatype AST fulfills two roles. First, it labels the system when used as the argument of PF. Second, the constructors of AST provide type representations on the value level, which can be used in generic functions to provide type-specific behaviour.

The pattern functor of AST is defined as follows:

```
type instance PF AST =
  K Int                :▷: Expr :+:   -- Const
  (I Expr :×: I Expr)  :▷: Expr :+:   -- Add
  (I Expr :×: I Expr)  :▷: Expr :+:   -- Mul
  I Var                :▷: Expr :+:   -- EVar
  (I Decl :×: I Expr)  :▷: Expr :+:   -- Let
  (I Var :×: I Expr)   :▷: Decl :+:   -- :=
  (I Decl :×: I Decl)  :▷: Decl :+:   -- Seq
  K String             :▷: Var        -- V
```

Note that the definition of PF AST is nearly the same as the definition of $\mathsf{PF}_{\mathsf{AST}}$ before. One difference is that the kinds of the components are now different.

We can now define the conversion functions per datatype. Since the actual implementations are as before, we only show the instance for Expr as an example:

```
instance Ix AST Expr where
  from (Const i)  = L                (Tag (K i))
  from (Add e e') = R (L             (Tag (ci e :×: ci e')))
  from (Mul e e') = R (R (L          (Tag (ci e :×: ci e'))))
  from (EVar x)   = R (R (R (L       (Tag (ci x)))))
  from (Let d e)  = R (R (R (R (L (Tag (ci d :×: ci e))))))
  to (L             (Tag (K i)))          = Const i
  to (R (L          (Tag (e :×: e'))))    = Add (di e) (di e')
  to (R (R (L       (Tag (e :×: e')))))   = Mul (di e) (di e')
  to (R (R (R (L    (Tag x)))))           = EVar (di x)
  to (R (R (R (R (L (Tag (d :×: e)))))))  = Let (di d) (di e)
  index = Expr
ci x = I (I* x)
di (I (I* x)) = x
```

The method *index* of class Ix provides access to the type representation of Expr on the value level. Such type representations will be useful later, when we define type-indexed functions.

The small GADT AST, the type family instance for PF and the instance definitions of Ix are the boilerplate code associated with our approach: the programmer has to provide this code for every system of datatypes. The boilerplate is regular enough so that it could be generated automatically, by building it into the compiler, using Template Haskell or a preprocessor, but it cannot be expressed directly within Haskell.

All the other code that we will cover now is generic for such systems of datatypes, so no further work is required in order to reap the fruits of our approach.

3.4 Recursion schemes

With the machinery introduced in Section 3.3, we can now define the recursion schemes from Section 3.2 for systems of mutually recursive types. Both *fold* and *compos* are based on *map*. Therefore, the key to defining generic recursion schemes is a generalization of *fmap* that we call *hmap*:

```
class HFunctor f where
  hmap :: (∀ix.Ix s ix ⇒ s ix → r ix → r' ix) → f s r ix → f s r' ix
```

This function is more general than *fmap* in two ways. First, the recursive structures being transformed (r and r′) are parametrized by an index. Second, the transforming function is polymorphic in the index type, and therefore *hmap* has a rank-2 type.

We define *hmap* by induction on the structure of the pattern functor. The only interesting instance is for I, where we apply the function parameter. In all the other instances, we just traverse the structure:

```
instance HFunctor (I xi) where
  hmap f (I x) = I (f index x)
instance HFunctor (K a) where
  hmap _ (K x) = K x
instance (HFunctor f, HFunctor g) ⇒ HFunctor (f :+: g) where
  hmap f (L x) = L (hmap f x)
  hmap f (R y) = R (hmap f y)
instance (HFunctor f, HFunctor g) ⇒ HFunctor (f :×: g) where
  hmap f (x :×: y) = hmap f x :×: hmap f y
instance HFunctor f ⇒ HFunctor (f :▷: ix) where
  hmap f (Tag x) = Tag (hmap f x)
```

3.4.1 Generic compos

Using *hmap*, it is easy to define *compos*:

```
compos :: (Ix s ix, HFunctor (PF s)) ⇒ (∀ix.Ix s ix ⇒ s ix → ix → ix) → ix → ix
compos f = to ∘ hmap (λix → I* ∘ f ix ∘ unI*) ∘ from
```

The only differences to the version in Section 3.2 are due to the presence of a type representation s ix and because the actual values in the structure are now wrapped in applications of the I_* constructor.

Bringert and Ranta (2006) describe in their paper on *compos* how to define the function on systems of mutually recursive datatypes. Their solution, however, requires to modify the system of datatypes and use a GADT representation such as the one in Section 3.3.2. Our version of *compos* works on systems of mutually recursive datatypes without modification. As an example, consider the following expression:

```
example = Let ("x" := Mul (Const 6) (Const 9))
              (Add (EVar "x") (EVar "y"))
```

The following function renames all variables in *example* – note how *renameVar'* can use the type representation to take different actions for different nodes – in this case, filter out nodes of type Var.

```
renameVar :: Expr → Expr
renameVar = renameVar' Expr
  where
    renameVar' :: Ix AST a ⇒ AST a → a → a
    renameVar' Var x = x ++ "_"
    renameVar' _   x = compos renameVar' x
```

The call *renameVar example* yields:

```
Let ("x_" := Mul (Const 6) (Const 9))
    (Add (EVar "x_") (EVar "y_"))
```

3.4.2 Generic fold

We can also define *fold* using *hmap*. Again, the definition is very similar to the single-datatype version:

```
type Algebra s r = ∀ix.Ix s ix ⇒ s ix → PF s s r ix → r ix
fold :: (Ix s ix, HFunctor (PF s)) ⇒ Algebra s r → ix → r ix
fold f = f index ∘ hmap (λ_ (I_* x) → fold f x) ∘ from
```

Using *fold* is slightly trickier than using *compos*, because we have to construct a suitable argument of type Algebra. This algebra argument involves a function operating on the pattern functor, which is itself a generically derived datatype.

We can facilitate the construction of algebras by defining suitable combinators:

```
(&) :: (a s r ix → r ix) → (b s r ix → r ix) →
       ((a :+: b) s r ix → r ix)
(f & g) (L x) = f x
(f & g) (R x) = g x
infixr 5 &
tag :: (a s r ix → r ix) → ((a :▷: ix) s r ix' → r ix')
tag f (Tag x) = f x
```

The $(\&)$ combinator lets us specify functions for different constructors separately, and *tag* is required to wrap tagged components. While it is possible to define more abbreviations for algebras, these two functions suffice to present an expression evaluator as an example.

Because different types in our system are mapped to different results, we need a family of datatypes for the result type of our algebra:

```
data family Value a :: *
data instance Value Expr = EV (Env → Int)
data instance Value Decl = DV (Env → Env)
data instance Value Var  = VV Var
type Env = [(Var, Int)]
```

An environment maps variables to integers. Expressions can contain variables, we therefore interpret them as functions from environments to integers. Declarations can be seen as environment transformers. Variables evaluate to their names. We can now state the algebra:

```
evalAlgebra :: Algebra AST Value
evalAlgebra _ =
    tag (λ(K x)                          → EV (const x))
  & tag (λ(I (EV x) :×: I (EV y))        → EV (λm → x m + y m))
  & tag (λ(I (EV x) :×: I (EV y))        → EV (λm → x m * y m))
  & tag (λ(I (VV x))                     → EV (fromJust ∘ lookup x))
  & tag (λ(I (DV e) :×: I (EV x))        → EV (λm → x (e m)))
  & tag (λ(I (VV x) :×: I (EV v))        → DV (λm → (x, v m) : m))
  & tag (λ(I (DV f) :×: I (DV g))        → DV (g ∘ f))
  & tag (λ(K x)                          → VV x)
```

Testing

```
eval :: Expr → Env → Int
eval x = let (EV f) = fold evalAlgebra x in f
```

in the expression *eval example* [("y", −12)] yields 42.

3.5 The Zipper

For a tree-like datatype, the Zipper (Huet 1997) is a derived data structure that allows efficient navigation through a tree, along its recursive nodes. At every moment, the Zipper keeps track of a *location*: a point of focus paired with a context that represents the rest of the tree. The focus can be moved up, down, left, and right.

For regular datatypes, it is well-known how to define Zippers generically (Hinze et al. 2004). In the following, we first show how to define a Zipper for a system of mutually recursive datatypes using our example of abstract syntax trees (Section 3.5.1). Then, in Section 3.5.2, we give a generic algorithm in terms of the representations derived in Section 3.3.

3.5.1 Zipper for mutually recursive datatypes

We first give a non-generic presentation of the Zipper for abstract syntax trees. We use the datatypes as given in Section 3.3.1.

A location is the current focus paired with context information. In a setting with multiple types, the type of the focus ix is not known – hence, we make it existential, and carry around a representation of type AST ix:

```
data Loc_AST :: ∗ → ∗ where
  Loc :: AST ix → ix → Ctxs_AST a ix → Loc_AST a
```

The type $\mathsf{Ctxs}_{\mathsf{AST}}$ encodes context information for the focus as a path from the focus to the root of the full tree. The path is stored in a stack of context frames:

```
data Ctxs_AST :: * → * → * where
  Empty :: Ctxs_AST a a
  (:.)  :: Ctx_AST ix b → Ctxs_AST a ix → Ctxs_AST a b
```

A context stack of type $\mathsf{Ctxs_{AST}\ a\ b}$ represents a value of type a with a b-typed *hole* in it. More specifically, a stack consists of frames of type $\mathsf{Ctx_{AST}\ ix\ b}$ that represent constructor applications that yield an ix-value with a hole of type b in it. The full tree that is represented by a location can be recovered by plugging the value in focus into the topmost context frame, plugging the resulting value into the next frame, and so on. For this to work, the target type ix of each context frame must be equal to the type of the hole in the remainder of the stack – as enforced by the type of $(:.)$.

Contexts

A single context frame $\mathsf{Ctx_{AST}}$ is following the structure of the types in the AST system closely.

```
data Ctx_AST :: * → * → * where
  AddC1  :: Expr → Ctx_AST Expr Expr
  AddC2  :: Expr → Ctx_AST Expr Expr
  MulC1  :: Expr → Ctx_AST Expr Expr
  MulC2  :: Expr → Ctx_AST Expr Expr
  EVarC  ::        Ctx_AST Expr Var
  LetC1  :: Expr → Ctx_AST Expr Decl
  LetC2  :: Decl → Ctx_AST Expr Expr
  BindC1 :: Expr → Ctx_AST Decl Var
  BindC2 :: Var  → Ctx_AST Decl Expr
  SeqC1  :: Decl → Ctx_AST Decl Decl
  SeqC2  :: Decl → Ctx_AST Decl Decl
```

The relation between $\mathsf{Ctx_{AST}}$ and AST becomes even more pronounced if we also look at the user-defined pattern functor $\mathsf{PF_{AST}}$ from Section 3.3.3. For every constructor in $\mathsf{PF_{AST}}$, we have as many constructors in $\mathsf{Ctx_{AST}}$ as there are recursive positions. Into a recursive position, we can descend. The type of the recursive position then becomes the second argument of $\mathsf{Ctx_{AST}}$. The other components of the original constructor are stored in the context. As an example, consider:

```
Let  ::  Decl →  Expr →          Expr
LetF :: r Decl → r Expr → PF_AST r Expr
```

We have two recursive positions. If we descend into the first, then Decl is the type of the hole, while Expr remains – and so we get

```
LetC1 :: Expr → Ctx_AST Expr Decl
```

If, however, we descend into the second position, then Expr is the type of the hole with Decl remaining:

$$\mathit{LetC2} :: \mathsf{Decl} \to \mathsf{Ctx}_{\mathsf{AST}}\ \mathsf{Expr}\ \mathsf{Expr}$$

Navigation

We now define functions that move the focus, transforming a location into a new location. These functions return their result in the Maybe monad, because navigation may fail: we cannot move down from a leaf of the tree, up from the root, or right if there are no more siblings in that direction.

Moving down analyzes the current focus. For all constructors that do not build leaves, we descend into the leftmost child by making it the new focus, and by pushing an appropriate frame onto the context stack. For leaves, we return *Nothing*.

$$\begin{aligned}
&\mathit{down} :: \mathsf{Loc}_{\mathsf{AST}}\ \mathsf{ix} \to \mathsf{Maybe}\ (\mathsf{Loc}_{\mathsf{AST}}\ \mathsf{ix})\\
&\mathit{down}\ (\mathit{Loc\ Expr}\ (\mathit{Add\ e\ e'})\ \mathit{cs}) = \mathit{Just}\ (\mathit{Loc\ Expr\ e}\ (\mathit{AddC1\ e'} :.\ \mathit{cs}))\\
&\mathit{down}\ (\mathit{Loc\ Expr}\ (\mathit{Mul\ e\ e'})\ \mathit{cs}) = \mathit{Just}\ (\mathit{Loc\ Expr\ e}\ (\mathit{MulC1\ e'} :.\ \mathit{cs}))\\
&\mathit{down}\ (\mathit{Loc\ Expr}\ (\mathit{EVar\ x})\ \ \mathit{cs}) = \mathit{Just}\ (\mathit{Loc\ Var}\ \ \mathit{x}\ (\mathit{EVarC}\ \ \ :.\ \mathit{cs}))\\
&\mathit{down}\ (\mathit{Loc\ Expr}\ (\mathit{Let\ d\ e})\ \ \mathit{cs}) = \mathit{Just}\ (\mathit{Loc\ Decl\ d}\ (\mathit{LetC1\ e}\ \ \ :.\ \mathit{cs}))\\
&\mathit{down}\ (\mathit{Loc\ Decl}\ (\mathit{x} := \mathit{e})\ \ \mathit{cs}) = \mathit{Just}\ (\mathit{Loc\ Var}\ \ \mathit{x}\ (\mathit{BindC1\ e} :.\ \mathit{cs}))\\
&\mathit{down}\ (\mathit{Loc\ Decl}\ (\mathit{Seq\ d\ d'})\ \mathit{cs}) = \mathit{Just}\ (\mathit{Loc\ Decl\ d}\ (\mathit{SeqC1\ d'} :.\ \mathit{cs}))\\
&\mathit{down}\ _ = \mathit{Nothing}
\end{aligned}$$

The function *up* is applicable whenever the current focus is not the root of the tree, i.e., whenever the context stack is non-empty. We then analyze the first context frame and plug in the current focus.

$$\begin{aligned}
&\mathit{up} :: \mathsf{Loc}_{\mathsf{AST}}\ \mathsf{ix} \to \mathsf{Maybe}\ (\mathsf{Loc}_{\mathsf{AST}}\ \mathsf{ix})\\
&\mathit{up}\ (\mathit{Loc}\ _\ \mathit{e}\ \ (\mathit{AddC1\ e'} :.\ \mathit{cs})) = \mathit{Just}\ (\mathit{Loc\ Expr}\ (\mathit{Add\ e\ e'})\ \mathit{cs})\\
&\mathit{up}\ (\mathit{Loc}\ _\ \mathit{e'}\ (\mathit{AddC2\ e}\ \ :.\ \mathit{cs})) = \mathit{Just}\ (\mathit{Loc\ Expr}\ (\mathit{Add\ e\ e'})\ \mathit{cs})\\
&\mathit{up}\ (\mathit{Loc}\ _\ \mathit{e}\ \ (\mathit{MulC1\ e'} :.\ \mathit{cs})) = \mathit{Just}\ (\mathit{Loc\ Expr}\ (\mathit{Mul\ e\ e'})\ \mathit{cs})\\
&\mathit{up}\ (\mathit{Loc}\ _\ \mathit{e'}\ (\mathit{MulC2\ e}\ \ :.\ \mathit{cs})) = \mathit{Just}\ (\mathit{Loc\ Expr}\ (\mathit{Mul\ e\ e'})\ \mathit{cs})\\
&\mathit{up}\ (\mathit{Loc}\ _\ \mathit{x}\ \ (\mathit{EVarC}\ \ \ :.\ \mathit{cs})) = \mathit{Just}\ (\mathit{Loc\ Expr}\ (\mathit{EVar\ x})\ \ \mathit{cs})\\
&\mathit{up}\ (\mathit{Loc}\ _\ \mathit{d}\ \ (\mathit{LetC1\ e}\ \ \ :.\ \mathit{cs})) = \mathit{Just}\ (\mathit{Loc\ Expr}\ (\mathit{Let\ d\ e})\ \ \mathit{cs})\\
&\mathit{up}\ (\mathit{Loc}\ _\ \mathit{e}\ \ (\mathit{LetC2\ d}\ \ \ :.\ \mathit{cs})) = \mathit{Just}\ (\mathit{Loc\ Expr}\ (\mathit{Let\ d\ e})\ \ \mathit{cs})\\
&\mathit{up}\ (\mathit{Loc}\ _\ \mathit{x}\ \ (\mathit{BindC1\ e} :.\ \mathit{cs})) = \mathit{Just}\ (\mathit{Loc\ Decl}\ (\mathit{x} := \mathit{e})\ \ \mathit{cs})\\
&\mathit{up}\ (\mathit{Loc}\ _\ \mathit{e}\ \ (\mathit{BindC2\ x} :.\ \mathit{cs})) = \mathit{Just}\ (\mathit{Loc\ Decl}\ (\mathit{x} := \mathit{e})\ \ \mathit{cs})\\
&\mathit{up}\ (\mathit{Loc}\ _\ \mathit{d}\ \ (\mathit{SeqC1\ d'} :.\ \mathit{cs})) = \mathit{Just}\ (\mathit{Loc\ Decl}\ (\mathit{Seq\ d\ d'})\ \mathit{cs})\\
&\mathit{up}\ (\mathit{Loc}\ _\ \mathit{d'}\ (\mathit{SeqC2\ d}\ \ :.\ \mathit{cs})) = \mathit{Just}\ (\mathit{Loc\ Decl}\ (\mathit{Seq\ d\ d'})\ \mathit{cs})\\
&\mathit{up}\ _ = \mathit{Nothing}
\end{aligned}$$

The function *right* succeeds for nodes that actually have a right sibling. The size of the context stack remains unchanged: we just replace its top element with a new frame.

```
right :: Loc_AST ix → Maybe (Loc_AST ix)
right (Loc _ e (AddC1 e' :. cs)) = Just (Loc Expr e' (AddC2 e  :. cs))
right (Loc _ e (MulC1 e' :. cs)) = Just (Loc Expr e' (MulC2 e  :. cs))
right (Loc _ d (LetC1 e  :. cs)) = Just (Loc Expr e  (LetC2 d  :. cs))
right (Loc _ x (BindC1 e :. cs)) = Just (Loc Expr e  (BindC2 x :. cs))
right (Loc _ d (SeqC1 d' :. cs)) = Just (Loc Decl d' (SeqC2 d  :. cs))
right _                          = Nothing
```

Using the Zipper

To use the Zipper, we need functions to turn syntax trees into locations, and back again. For manipulating trees, we provide an update operation that replaces the subtree in focus.

To enter the tree, we place it into the empty context:

```
enter :: Expr → Loc_AST Expr
enter e = Loc Expr e Empty
```

To leave, we move up as far as possible and then return the expression in focus.

```
leave :: Loc_AST Expr → Expr
leave (Loc _ e Empty) = e
leave loc             = leave (fromJust (up loc))
```

To update the tree, we pass in a function capable of modifying the current point of focus. Because the value in focus can have different types, this function needs to be parameterized by the type representation.

```
update :: (∀ix.AST ix → ix → ix) →
          Loc_AST Expr → Loc_AST Expr
update f (Loc ix x cs) = Loc ix (f ix x) cs
```

As an example, we modify the multiplication in

```
example = Let ("x" := Mul (Const 6) (Const 9))
              (Add (EVar "x") (EVar "y"))
```

To combine the navigation and edit operations, it is helpful to make use of flipped function composition $(\ggg) :: (a \to b) \to (b \to c) \to (a \to c)$ and monadic composition $(>=>) :: \text{Monad } m \Rightarrow (a \to m\ b) \to (b \to m\ c) \to (a \to m\ c)$. The call

```
enter ⋙ down >=> down >=> right >=> update solve ⋙
leave ⋙ return $ example
```

with

```
solve :: AST ix → ix → ix
solve Expr _ = Const 42
solve _    x = x
```

results in

```
Just (Let ("x" := Const 42) (Add (EVar "x") (EVar "y")))
```

3.5.2 A generic Zipper

We now define a Zipper generically for a system of mutually recursive datatypes. We make the same steps as in the example for abstract syntax trees before.

The type definitions for locations and context stacks stay essentially the same:

```
data Loc :: (∗ → ∗) → ∗ → ∗ where
  Loc :: (Ix s ix, Zipper (PF s)) ⇒ ix → Ctxs s a ix → Loc s a
data Ctxs :: (∗ → ∗) → ∗ → ∗ → ∗ where
  Empty :: Ctxs s a a
  (:.) :: Ix s ix ⇒ Ctx (PF s) s ix b → Ctxs s a ix → Ctxs s a b
```

Instead of storing a type representation in a Loc such as AST ix, we now require two things via class constraints: the type ix must be part of the system s, as expressed by Ix s ix. Furthermore, we need a Zipper for the system s. This condition is expressed by Zipper (PF s) and will be explained in more detail below.

In the stack Ctxs, we also require that the types of the elements are in s via the constraint Ix s ix.

Contexts

The context type is defined generically on the pattern functor of s. We thus reuse the type family PF defined in Section 3.3. We have to distinguish between different type constructors that make up the pattern functor, and therefore define Ctx as a datatype family:

```
data family Ctx f :: (∗ → ∗)   -- datatype system s
                  → ∗          -- index type ix
                  → ∗          -- hole type b
                  → ∗
```

The simple cases are for constant types, sums and products. There is a correspondence between the context of a datatype and its formal derivative (McBride 2001):

```
data instance Ctx (K a)     s ix b = CK Void
data instance Ctx (f :+: g) s ix b = CL (Ctx f s ix b)
                                   | CR (Ctx g s ix b)
data instance Ctx (f :×: g) s ix b = C1 (Ctx f s ix b) (g s I∗ ix)
                                   | C2 (f s I∗ ix) (Ctx g s ix b)
```

For constants, there are no recursive positions, hence we produce an empty datatype, i.e., a datatype with no constructors:

```
data Void
```

For a sum, we are given either an f or a g, and compute the context of that. For a product, we can descend either left or right. If we descend into f, we pair a context for f with g. If we descend into g, we pair f with a context for g.

We are left with the cases for I and (:▷:). According to the analogy with the derivative, the context of the identity should be the unit type. However, we are in a situation where there are multiple types involved. The type index of I fixes the type of the hole. We express this type equality as follows, by means of a GADT:[1]

```
data instance Ctx (I xi) s ix b where
  CId :: Ctx (I xi) s ix xi
```

For the case of tags, we have a similar situation. A tag does not affect the structure of the context, it only provides information for the type system. In this case, not the type of the hole, but the type of the context itself is required to match the type index of the tag:

```
data instance Ctx (f :▷: xi) s ix b where
  CTag :: Ctx f s xi b → Ctx (f :▷: xi) s xi b
```

This completes the definition of Ctx. We can convince ourselves that instantiating Ctx to PF AST results in a datatype that is isomorphic to $\mathsf{Ctx}_{\mathsf{AST}}$. It is also quite a bit more complex than the hand-written variant, but fortunately, the programmer never has to use it directly. Instead, we can interface with it using generic navigation functions.

Navigation

The navigation functions are again generically defined on the structure of the pattern functor. Thus, we define them in a class Zipper:

```
class Zipper f where
  ...
```

We will fill this class with methods incrementally.

Down To move down in a tree, we define a generic function *first* in our class Zipper:

```
class Zipper f where
  ...
  first :: (∀b.Ix s b ⇒ b → Ctx f s ix b → a) → f s I* ix → Maybe a
```

[1] Currently, GHC does not allow instances of datatype families to be defined as GADTs. In the actual implementation, we therefore simulate the GADT by including an explicit proof of type equality (Peyton Jones et al. 2006; Baars and Swierstra 2002).

The function takes a functor $\mathsf{f\ s\ I}_*\ \mathsf{ix}$ and tries to split off its first recursive component. This is of some type b with $\mathsf{Ix\ s\ b}$. The rest is a context of type $\mathsf{Ctx\ f\ s\ ix\ b}$. The function takes a continuation parameter that describes what to do with the two parts. Function *down* is defined in terms of *first*:

$$\begin{aligned}&\mathit{down} :: \mathsf{Loc\ s\ ix} \to \mathsf{Maybe\ (Loc\ s\ ix)}\\&\mathit{down}\ (\mathit{Loc}\ x\ cs) = \mathit{first}\ (\lambda z\ c \to \mathit{Loc}\ z\ (c :.\ cs))\ (\mathit{from}\ x)\end{aligned}$$

We try to split the tree in focus x. If this succeeds, we get a new focus z and a new context frame c. We push c on the stack.

We define *first* by induction on the structure of pattern functors. Constant types constitute the leaves in the tree. We cannot descend, and return *Nothing*.

$$\begin{aligned}&\textbf{instance}\ \mathsf{Zipper\ (K\ a)}\ \textbf{where}\\&\quad \ldots\\&\quad \mathit{first}\ f\ (K\ a) = \mathit{Nothing}\end{aligned}$$

In a sum, we descend further, and add the corresponding context constructor *CL* or *CR* to the context.

$$\begin{aligned}&\textbf{instance}\ (\mathsf{Zipper\ f}, \mathsf{Zipper\ g}) \Rightarrow \mathsf{Zipper\ (f :+: g)}\ \textbf{where}\\&\quad \ldots\\&\quad \mathit{first}\ f\ (L\ x) = \mathit{first}\ (\lambda z\ c \to f\ z\ (\mathit{CL}\ c))\ x\\&\quad \mathit{first}\ f\ (R\ y) = \mathit{first}\ (\lambda z\ c \to f\ z\ (\mathit{CR}\ c))\ y\end{aligned}$$

We want to get to the first child. Therefore, we first try to descend to the left in a product. Only if that fails (*mplus*), we try to split the right component.

$$\begin{aligned}&\textbf{instance}\ (\mathsf{Zipper\ f}, \mathsf{Zipper\ g}) \Rightarrow \mathsf{Zipper\ (f :\times: g)}\ \textbf{where}\\&\quad \ldots\\&\quad \mathit{first}\ f\ (x :\times: y) = \mathit{first}\ (\lambda z\ c \to f\ z\ (\mathit{C1}\ c\ y))\ x\ \text{`}\mathit{mplus}\text{`}\\&\qquad\qquad\qquad\quad \mathit{first}\ (\lambda z\ c \to f\ z\ (\mathit{C2}\ x\ c))\ y\end{aligned}$$

In the I case, we have exactly one possibility. We split $I\ (I_*\ x)$ into x and the context *CId* and pass the two parts to the continuation f:

$$\begin{aligned}&\textbf{instance}\ \mathsf{Zipper\ (I\ xi)}\ \textbf{where}\\&\quad \ldots\\&\quad \mathit{first}\ f\ (I\ (I_*\ x)) = \mathit{return}\ (f\ x\ \mathit{CId})\end{aligned}$$

It is interesting to see why this types: the type of x is xi, so applying f to x instantiates b to xi and forces the second argument of f to be of type $\mathsf{Ctx\ (I\ xi)\ s\ ix\ xi}$. But that is exactly the type of *CId*.

Finally, for a tag, we also descend further and apply *CTag* to the context.

$$\begin{aligned}&\textbf{instance}\ \mathsf{Zipper\ f} \Rightarrow \mathsf{Zipper\ (f :\triangleright: xi)}\ \textbf{where}\\&\quad \ldots\\&\quad \mathit{first}\ f\ (\mathit{Tag}\ x) = \mathit{first}\ (\lambda z\ c \to f\ z\ (\mathit{CTag}\ c))\ x\end{aligned}$$

This types because *Tag* introduces the refinement that *CTag* requires: applying *CTag* to *c* results in Ctx (f :▷: xi) s xi b. This can be passed to *f* only if ix from the type of *first* is equal to xi. But it is, because the pattern match on *Tag* forces it to be.

Up Now that we can move down, we also want to move up again. We employ the same scheme as before: using an inductively defined generic helper function *fill*, we then define *up*. The function *fill* has the following type:

```
class Zipper f where
  ...
  fill :: Ix s b ⇒ b → Ctx f s ix b → f s I* ix
```

The function takes a value together with a compatible context frame and plugs them together, producing a value of the pattern functor. This operation is total, so no Maybe is required in the result.

With *fill*, we can define *up* as follows:

```
up :: Loc s ix → Maybe (Loc s ix)
up (Loc x Empty)   = Nothing
up (Loc x (c :. cs)) = Just (Loc (to (fill x c)) cs)
```

We cannot move up in the root of the tree and thus fail on an empty context stack. Otherwise, we pick the topmost context frame, and call *fill*. Since *fill* results in a value of the pattern functor, we have to convert back into the original form using *to*.

We give the instances for *fill*, starting with K. As an argument to *fill*, we need a context for K, for which we defined but one constructor *CK* with a Void parameter. In other words, in order to call *fill* on K, we have to produce a value of Void, which, apart from ⊥, is impossible. In the context of our Zipper library, we can guarantee that ⊥ is never produced for Void. We therefore define:

```
instance Zipper (K a) where
  ...
  fill x (CK void) = impossible void
impossible :: Void → a
impossible void = error "impossible"
```

Nothing interesting happens in the sum case. We simply call *fill* recursively on the branch we are in:

```
instance (Zipper f, Zipper g) ⇒ Zipper (f :+: g) where
  ...
  fill x (CL c) = L (fill x c)
  fill x (CR c) = R (fill x c)
```

For products we also fill recursively, on the argument indicated by the context:

```
instance (Zipper f, Zipper g) ⇒ Zipper (f :×: g) where
  ...
  fill x (C1 c y) = fill x c :×: y
  fill y (C2 x c) = x :×: fill y c
```

For I, we return the element to plug itself, wrapped by the appropriate constructors:

```
instance Zipper (I xi) where
  ...
  fill x CId = I (I* x)
```

Again, this only types because of the refinement introduced by *CId*: the x is of type b, so $I\ (I_*\ x)$ would normally be of type I b s I_* ix, not I xi s I_* ix. But pattern matching on *CId* forces b and xi to be equal.

For a tag, we call *fill* recursively on the tagged context:

```
instance Zipper f ⇒ Zipper (f :▷: xi) where
  ...
  fill x (CTag c) = Tag (fill x c)
```

Once more, the refinement introduced by *CTag* is required for the use of *Tag* to be correct.

Right As a final example of a navigation function, we define *right*. We again employ the same scheme as before. We define a generic function *next* with the following type:

```
class Zipper f where
  ...
  next :: (∀b.Ix s b ⇒ b → Ctx f s ix b → a) →
          (Ix s b ⇒ b → Ctx f s ix b → Maybe a)
```

The function takes a context frame and an element that fits into the context. By looking at the context, it tries to move the focus one element to the right, thereby producing a new element – possibly of different type – and a new compatible context. These can, as in *first*, be combined using the passed continuation.

With *next*, we can define *right*:

```
right :: Loc s ix → Maybe (Loc s ix)
right (Loc x Empty)    = Nothing
right (Loc x (c :. cs)) = next (λz c' → Loc z (c' :. cs)) x c
```

We cannot move right in the root of the tree, thus *right* fails in an empty context. Otherwise, we only need to look at the topmost context frame, and pass it to *next*, together with the current focus. On success, we take the new focus, and push the new context frame back on the stack.

The case *next* for K is again impossible:

```
instance Zipper (K a) where
  ...
  next f x (CK void) = impossible void
```

In the case for sums we just call *next* recursively:

```
instance (Zipper f, Zipper g) ⇒ Zipper (f :+: g) where
  ...
  next f x (CL c) = next (λz c → f z (CL c)) x c
  next f y (CR c) = next (λz c → f z (CR c)) y c
```

The case for products is the most interesting one. If we are currently in the first component, we try to move to the next element there, but if this fails, we have to select the first child of the second component, calling *first*. In that case, we also have to plug the old focus x back into its context c, using *fill*. If, however, we are already in the right component, we do not need a case distinction and just try to move further to the right using *next*.

```
instance (Zipper f, Zipper g) ⇒ Zipper (f :×: g) where
  ...
  next f x (C1 c y) = next (λz c' → f z (C1 c'       y )) x c `mplus`
                      first (λz c' → f z (C2 (fill x c) c')) y
  next f y (C2 x c) = next (λz c' → f z (C2 x       c')) y c
```

Since I represents a single child, we cannot move right in such a location:

```
instance Zipper (I xi) where
  ...
  next f x CId = Nothing
```

On a tagged type, we recurse:

```
instance Zipper f ⇒ Zipper (f :▷: xi) where
  ...
  next f x (CTag c) = next (λz c' → f z (CTag c')) x c
```

Using the Zipper

The functions *enter*, *leave* and *update* can be converted from the specific case for AST almost without change. We have to adapt the types and respect the fact that *Loc* no longer carries a type representation. Instead, we must add a type representation as an argument to *enter* to help the type checker to associate a system of datatypes s with the type ix.

```
enter :: (Ix s ix, Zipper (PF s)) ⇒ s ix → ix → Loc s ix
enter _ x = Loc x Empty
leave :: Loc s ix → ix
leave (Loc x Empty) = x
leave loc           = leave (fromJust (up loc))
update :: (∀ix.Ix s ix ⇒ s ix → ix → ix) → Loc s ix → Loc s ix
update f (Loc x cs) = Loc (f index x) cs
```

Let us repeat the example from before, but now use the generic Zipper: apart from the additional argument to *enter*, nothing changes

```
enter Expr >>> down >>> down >>> right >>> update solve >>>
leave >>> return $ example
```

and the result is also the same:

```
Just (Let ("x" := Const 42) (Add (EVar "x") (EVar "y")))
```

3.6 Generic rewriting

Term rewriting can be specified generically, for arbitrary regular datatypes, if these are viewed as fixed points of functors (Jansson and Jeuring 2000; Van Noort et al. 2008). In the following we show how to generalize term rewriting even further, to work on systems with an arbitrary number of datatypes. For reasons of space, we do not discuss generic rewriting in complete detail, but focus on the operation of matching the left-hand side of a rule with a term.

3.6.1 Schemes of regular datatypes

Before tackling matching on systems of mutually recursive datatypes, we briefly sketch the ideas behind its implementation on regular datatypes. Consider how to implement matching for the simple version of the Expr datatype introduced in Section 3.2. First, we define expression schemes, which extend expressions with a constructor for rule meta-variables. Then we define matching of those schemes against expressions:

```
data ExprS = MetaVar String
           | ConstS  Int
           | AddS    ExprS ExprS
           | MulS    ExprS ExprS
match :: ExprS → Expr → Maybe [(String, Expr)]
```

On success, *match* returns a substitution mapping meta-variables to matched subterms. For example, the call

$$match\ (MulS\ (MetaVar\ \texttt{"x"})\ (MetaVar\ \texttt{"y"}))\\ \quad (Mul\ (Const\ 6)\ (Const\ 9))$$

yields $Just\ [(\texttt{"x"}, Const\ 6), (\texttt{"y"}, Const\ 9)]$.

To implement *match* generically, we need to define the scheme of a datatype generically. To this end, recall that a regular datatype is isomorphic to the type Fix f, for a suitably defined f. A meta-variable can appear deep inside a scheme, this suggests that the extension with *MetaVar* should take place inside the recursion, and hence on f. This motivates the following definition for schemes of regular datatypes:

$$\textbf{type}\ \mathsf{Scheme\ a} = \mathsf{Fix\ (K\ String :+: PF\ a)}$$

For example, the expression scheme that is used above as the first argument to *match* can be represented by

$$In\ (R\ (R\ (R\ (I\ (In\ (L\ (K\ \texttt{"x"})))) :\times: I\ (In\ (L\ (K\ \texttt{"y"}))))))$$

3.6.2 Schemes of a datatype system and substitutions

A system of mutually recursive datatypes requires as many sorts of meta-variables as there are datatypes. For example, for the system used in Section 3.3, we need three meta-variables, ranging over Expr, Decl and Var, respectively. Fortunately, we can deal with all these meta-variables in one go:

$$\textbf{type}\ \mathsf{Scheme\ s} = \mathsf{HFix\ ((K\ String :+: PF\ s)\ s)}$$

As in the regular case, the pattern functor is extended with a meta-variable representation. We want meta-variable representations to be polymorphic, so, unlike other constructors, K String is not tagged with $(:\triangleright:)$. Now, the same representation can be used to encode meta-variables that match, for example, Expr, Decl and Var.

Dealing with multiple datatypes affects the types of substitutions. We cannot use a homogeneous list of mappings as we did earlier, because different meta-variables may map to different datatypes. We get around this difficulty by existentially quantifying over the type of the matched datatype:

$$\textbf{data}\ \mathsf{DynIx\ s} = \forall \mathsf{ix.Ix\ s\ ix} \Rightarrow \mathsf{DynIx\ (s\ ix)\ ix}\\ \textbf{type}\ \mathsf{Subst\ s} = [(\mathsf{String}, \mathsf{DynIx\ s})]$$

3.6.3 Generic matching

Generic matching is defined as follows[2]:

[2] Currently, GHC does not unify the types $\mathsf{PF\ s\ s\ ix}$ and $\mathsf{PF\ s'\ s'\ ix}$, even if it unifies the equivalent types that use equality constraints. This problem causes GHC 6.8.3 to reject *matchM*. To make *matchM* typeable, the actual implementation desugars the type of *from* to $(\mathsf{Ix\ s\ ix}, \mathsf{a} \sim \mathsf{PF\ s}) \Rightarrow \mathsf{ix} \rightarrow \mathsf{a\ s\ I_*\ ix}$.

```
type MatchM s a = StateT (Subst s) Maybe a
matchM :: (HZip (PF s), Ix s ix) ⇒ Scheme s ix → I* ix → MatchM s ()
matchM (HIn (L (K metavar))) (I* e)
  = do subst ← get
       case lookup metavar subst of
         Nothing → put ((metavar, DynIx index e) : subst)
         Just _  → fail ("repeated use: " ++ metavar)
matchM (HIn (R r)) (I* e)
  = combine matchM r (from e)
```

Generic matching tries to match a term (I_* ix) against a scheme of that type (Scheme s ix). The resulting information is returned in the MatchM monad. The definition of MatchM uses Maybe for indicating possible failure, and on top of that monad we use the state transformer StateT. The state monad is used to thread the substitution as we traverse the scheme and the term in parallel.

Generic matching consists of two cases. When dealing with a meta-variable, we first check that there is no previous mapping for it. (For the sake of brevity, we do not show how to deal with multiple occurrences of a meta-variable.) If that is the case, we update the state with the new mapping.

The second case deals with matching constructors against constructors. More specifically, this corresponds to matching the term *Mul* (*Const* 6) (*Const* 9) against the scheme *MulS* (*MetaVar* `"x"`) (*MetaVar* `"y"`). This is handled by the generic function *combine*, which matches the two pattern functor representations. If the representations match (as in our example), then *matchM* is applied to the recursive occurrences (for instance, on *MetaVar* `"x"` and *Const* 6, and *MetaVar* `"y"` and *Const* 9).

Now we can write the following wrapper on *matchM* to hide the use of the state monad that threads the substitution:

```
match :: (HZip (PF s), Ix s ix) ⇒ Scheme s ix → ix → Maybe (Subst s)
match scheme tm = execStateT (matchM scheme (I* tm)) []
```

3.6.4 Generic zip and combine

The generic function *combine* is defined in terms of another function, which is a generalization of *zipWith* for arbitrary functors. Like *hmap*, the function *hzipM* is defined by induction on the pattern functor by means of a type class:

```
class HZip f where
  hzipM :: Monad m ⇒
           (∀ix.Ix s ix ⇒ s ix → r ix → r' ix → m (r'' ix)) →
           f s r ix → f s r' ix → m (f s r'' ix)
```

The function *hzipM* takes an argument that combines the r and r′ structures stored in the pattern functor. The traversal is performed in a monad to notify failure when the functor arguments do not match, and to allow the argument to use state, for example.

In our case, we are not interested in the resulting merged structure ($\mathsf{r}''\ \mathsf{ix}$). Indeed, *matchM* stores information only in the state monad, so we define *combine* to ignore the result.

```
data K* a b = K*{unK* :: a}
combine :: (Monad m, HZip f) ⇒
           (∀ix.Ix s ix ⇒ r ix → r' ix → m ()) →
           f s r ix → f s r' ix → m ()
combine f x y = do hzipM wrapf x y
                   return ()
  where wrapf _ x y = do f x y
                         return (K* ())
```

In the above, K_* is used to ignore the type ix in the result. The definition of *hzipM* does not differ much from that used when dealing with a single regular datatype:

```
instance HZip (I xi) where
  hzipM f (I x) (I y) = liftM I (f index x y)
instance (HZip a, HZip b) ⇒ HZip (a :×: b) where
  hzipM f (x1 :×: x2) (y1 :×: y2) = liftM2 (:×:) (hzipM f x1 y1) (hzipM f x2 y2)
instance (HZip a, HZip b) ⇒ HZip (a :+: b) where
  hzipM f (L x) (L y) = liftM L (hzipM f x y)
  hzipM f (R x) (R y) = liftM R (hzipM f x y)
  hzipM f _     _     = fail "zip failed in :+:"
instance HZip f ⇒ HZip (f :▷: ix) where
  hzipM f (Tag x) (Tag y) = liftM Tag (hzipM f x y)
instance Eq a ⇒ HZip (K a) where
  hzipM f (K x) (K y) | x ≡ y       = return (K x)
                      | otherwise = fail "zip failed in K"
```

In the definition above, we use *liftM* and *liftM2* to turn the pure structure constructors into monadic functions.

3.7 Related work

Malcolm (1990) shows how to define two mutually recursive types as initial objects of functor-algebras. Swierstra et al. (1999) show how to implement fixed points for mutual recursive datatypes in Haskell. They introduce a new fixed point for every arity of mutually recursive datatypes. None of these approaches can be used as a basis for an implementation of fixed points for mutually recursive datatypes in Haskell suitable for implementing generic programs.

Several authors discuss how to generate folds and other recursive schemes on mutually recursive datatypes (Böhm and Berarducci 1985; Sheard and Fegaras 1993; Swierstra et al. 1999; Lämmel et al. 2000). The definitions in these papers cannot be directly

translated to Haskell because they require (type level) induction on the number of datatypes involved.

Mitchell and Runciman (2007b) show how to obtain traversals for mutually recursive datatypes using the class Biplate. However, the type on which an action is performed remains fixed during a traversal. In contrast, the recursion schemes from Section 3.4 can apply their function arguments to subtrees of different types.

Since dependently typed programming languages have a much more powerful type system than Haskell extended with GADTs and type families, it is possible to define fixed-points for mutually recursive datatypes in many dependently typed programming languages. Benke et al. (2003) give a formal construction for mutually recursive datatypes as indexed inductive definitions in Alfa. Some similarities with our work are that the pattern functor argument is indexed by the datatype sort, and recursive positions specify the sort index of the subtree. Altenkirch and McBride (2003) show how to do generic programming in the dependently typed programming language OLEG. We believe that it is easier to write generic programs on mutually recursive datatypes in our approach, since we do not haves to deal with kind-indexed definitions, environments, type applications, datatype variables and argument variables, in addition to the cases for sums, products and constants.

McBride (2001) first described a generic Zipper on regular datatypes, which was implemented in Epigram by Morris et al. (2006). The Zipper has been used as an example of a type-indexed datatype in Generic Haskell (Hinze et al. 2004), but again only for regular datatypes. The dissection operator introduced by McBride (2008) is also only defined for regular datatypes, although McBride remarks that an implementation in a dependently typed programming language for mutually recursive datatypes is possible.

3.8 Conclusions

Until now, many powerful generic algorithms were known, but their adoption in practice has been hindered by their restriction to regular datatypes. In this chapter, we have shown that we can overcome this restriction in a way that is directly applicable in practice: using recent extensions of Haskell, we can define generic programs that exploit the recursive structure of datatypes on systems of arbitrarily many mutually recursive datatypes. For instance, extensive use of generic programming becomes finally feasible for compilers, which are often based on an abstract syntax that consists of many mutually recursive datatypes. Furthermore, our approach is non-invasive: the definitions of large systems of datatypes need not be modified in order to use generic programming.

Furthermore, we have demonstrated our approach by implementing several recursion schemes such as *compos* and *fold*, the Zipper, and rewriting functionality.

The code for this chapter is available and will be released as a Haskell library soon.

In the future, we hope to investigate the application of our representation using $(:\triangleright:)$ to arbitrary GADTs, hopefully giving us *fold* and other generic operations on GADTs, similar to the work of Johann and Ghani (2008).

Acknowledgements José Pedro Magalhães and Marcos Viera commented on a previous paper version of this chapter. Claus Reinke suggested to us the "type families desugaring trick" when we had trouble getting our code type correct. This research has been partially funded by the Netherlands Organisation for Scientific Research (NWO), through its projects on "Real-life Datatype-Generic Programming" (612.063.613) and "Scriptable Compilers" (612.063.406).

4 Comparing Libraries for Generic Programming in Haskell

Datatype-generic programming is defining functions that depend on the structure, or "shape", of datatypes. It has been around for more than 10 years, and a lot of progress has been made, in particular in the lazy functional programming language Haskell. There are more than 10 proposals for generic programming libraries or language extensions for Haskell.

In this chapter we compare and characterise the many generic programming libraries for Haskell. To that end, we introduce a set of criteria and develop a generic programming benchmark: a set of characteristic examples testing various facets of datatype-generic programming. We have implemented the benchmark for ten existing Haskell generic programming libraries and present the evaluation of the libraries. The comparison is useful for reaching a common standard for generic programming, but also for a programmer who has to choose a particular approach for datatype-generic programming.

4.1 Introduction

Software development often consists of designing a datatype to which functionality is added. Some functionality is datatype specific. Other functionality is defined on almost all datatypes, and only depends on the structure of the datatype; this is called datatype-generic functionality. Examples of such functionality are comparing two values for equality, searching a value of a datatype for occurrences of a particular string or other value, editing a value, pretty-printing a value, etc. Larger examples include XML tools, testing frameworks, debuggers, and data-conversion tools.

Datatype-generic programming has been around for more than 10 years now. A lot of progress has been made in the last decade, in particular with generic programming in Haskell. There are more than 10 proposals for generic programming libraries or language extensions for Haskell. Such libraries and extensions are also starting to appear for other programming languages, such as ML.

Although generic programming has been used in several applications, it has few users for real-life projects. This is understandable. Developing a large application takes a couple of years, and choosing a particular approach to generic programming for such a project involves a risk. Few approaches that have been developed over the last decade are still supported, and there is a high risk that the chosen approach will not be

supported anymore, or that it will change in a backwards-incompatible way in a couple of years time.

The Haskell Refactorer HaRe (Li et al. 2003) is an exception, and provides an example of a real-life project in which a generic-programming technique – Strafunski (Lämmel and Visser 2002) – is used to implement traversals over a large abstract syntax tree. However, this project contains several other components that could have been implemented using generic-programming techniques, such as rewriting, unification, and pretty-printing modules. These components are much harder to implement than traversals over abstract-syntax trees. Had these components been implemented generically, we claim that, for example, the recent work of the HaRe team to adapt the refactoring framework to the Erlang language (Derrick and Thompson 2005) would have been easier. Other projects that use generic programming are the Haskell Application Server (HAppS), which uses the extensible variant of Scrap Your Boilerplate, and the Catch and Reach tools (Mitchell and Runciman 2007a; Naylor and Runciman 2007), which use the Uniplate library to implement traversals.

It is often not immediately clear which generic programming approach is best suited for a particular project. There are generic functions that are difficult or impossible to define in certain approaches. The datatypes to which a generic function can be applied, and the amount of work a programmer has to do per datatype and/or generic function varies among different approaches.

The current status of generic programming in Haskell is comparable to the lazy Tower of Babel preceding the birth of Haskell in the eighties (Hudak et al. 2007). We have many single-site languages or libraries, each individually lacking critical mass in terms of language/library-design effort, implementations, and users.

How can we decrease the risk in using generic programming? Our eventual goal is to design a *common generic programming library* for Haskell. To increase the chances of continuing support, we would develop this library in an international committee. The rationale for developing a library for generic programming instead of a language extension is that Haskell is powerful enough to write generic programs that previously needed the support of language extensions such as PolyP (Jansson and Jeuring 1997) or Generic Haskell (Löh et al. 2003). Furthermore, compared with a language extension, a library is much easier to ship, support, and maintain. The library might be accompanied by tools that depend on non-standard language extensions, for example for generating embedding-projection pairs, as long as the core is standard Haskell. The standard that the library design should target is Haskell 98 and widely-accepted extensions (such as existential types and multi-parameter type classes) that are likely to be included in the next Haskell standard (Peyton Jones et al. 2007). The library should support the most common generic programming scenarios, so that programmers can define the generic functions that they want and use them with the datatypes they want.

To design a common generic programming library, we first have to evaluate existing libraries to find out differences and commonalities, and to be able to make well-motivated decisions about including and excluding features. In this chapter we take the first step towards our goal. We design a framework to compare generic programming libraries in an expressive functional programming language, and apply this framework

to Haskell. We will evaluate and compare the following libraries:

- Lightweight Implementation of Generics and Dynamics (LIGD) (Cheney and Hinze 2002)
- Polytypic programming in Haskell (PolyLib) (Norell and Jansson 2004)
- Scrap your boilerplate (SYB) (Lämmel and Peyton Jones 2003, 2004)
- Scrap your boilerplate, extensible variant using type classes (SYB3) (Lämmel and Peyton Jones 2005)
- Scrap your boilerplate, spine view variant (Spine) (Hinze et al. 2006; Hinze and Löh 2006)
- Extensible and Modular Generics for the Masses (EMGM) (Oliveira et al. 2006) based on (Hinze 2006).
- RepLib: a library for derivable type classes (Weirich 2006)
- Smash your boilerplate (Smash) (Kiselyov 2006)
- Uniplate (Mitchell and Runciman 2007b)
- MultiRec: a library that supports fixed-point views for mutually recursive datatypes. MultiRec was introduced in Chapter 3.

Note that this list does not contain generic programming language extensions such as PolyP and Generic Haskell, and no pre-processing approaches to generic programming such as DrIFT (Winstanley and Meacham 2006) and Data.Derive. We strictly limit ourselves to library approaches, which, however, might be based on particular compiler extensions. The SYB and Strafunski (Lämmel and Visser 2003) approaches are very similar, and therefore we only take the SYB approach into account in this evaluation. The functionality of the Compos library (Bringert and Ranta 2006) is subsumed by Uniplate, and hence we only evaluate the latter.

We evaluate existing libraries by means of a set of criteria. Papers about generic programming usually give desirable criteria for generic programs. Examples of such criteria are: can a generic function be extended with special behaviour on a particular datatype, and are generic functions first-class, that is, can they take a generic function as argument. We develop a set of criteria based on our own ideas about generic programming, and ideas from papers about generic programming. For most criteria, we have a generic test function that determines whether or not the criterion is satisfied. These test functions together form a benchmark which we try to implement for the different approaches.

We are aware of three existing comparisons of support for generic programming in programming languages. Garcia et al. (2007) and Bernardy et al. (2008) compare the support for *property-based* generic programming across different programming languages. Haskell type classes support all the eight criteria of Garcia et al. We use more

fine-grained criteria to distinguish the Haskell libraries support for *datatype*-generic programming. Hinze et al. (2007) compare various approaches to datatype-generic programming in Haskell. However, most of the covered approaches are language extensions, and many of the recent library approaches have not been included.

This chapter has the following contributions:

- It gives an extensive set of *criteria for comparing libraries for generic programming in Haskell*. The criteria might be viewed as a characterisation of generic programming in Haskell.
- It develops a *generic programming benchmark*: a set of characteristic examples with which we can test the criteria for generic programming libraries.
- It compares ten existing library approaches to generic programming in Haskell with respect to the criteria, using the implementation of the benchmark in the different libraries.
- The benchmark itself is a contribution. It can be seen as a cookbook that illustrates how different generic programming tasks are achieved using the different approaches. Furthermore, its availability makes it easier to compare the expressiveness of future generic programming libraries. The benchmark suite can be obtained following the instructions at `http://haskell.org/haskellwiki/GPBench`.

The outcome of this evaluation is not necessarily restricted to the context of Haskell. We think this comparison will be relevant for other programming languages as well. This chapter will be useful for a programmer that develops a generic programming library in Haskell, and for a programmer with knowledge of the concepts behind generic programming that wants to select a library for a particular purpose, which requires generic programming techniques. We assume the reader is familiar with generic programming.

This chapter is organized as follows. Section 4.2 introduces datatype-generic programming concepts and terminology. Section 4.3 shows the design and contents of the benchmark suite. Section 4.4 introduces and discusses the criteria we use for comparing libraries for generic programming in Haskell. Section 4.5 summarizes the evaluation of the different libraries with respect to the criteria, using the benchmark. Section 4.6 gives an overview of each library compared in this chapter. Section 4.7 presents the evaluation in full detail. Section 4.8 concludes.

4.2 Generic programming: concepts and terminology

This section introduces and illustrates generic programming using a simplified form of the datatype-generic programming library LIGD. We use LIGD because the encoding

```
geq :: Rep a → a → a → Bool
geq (RUnit)          Unit          Unit      = True
geq (RSum r_a r_b)   (Inl a_1)     (Inl a_2) = geq r_a a_1 a_2
geq (RSum r_a r_b)   (Inr b_1)     (Inr b_2) = geq r_b b_1 b_2
geq (RSum r_a r_b)   _             _         = False
geq (RProd r_a r_b) (Prod a_1 b_1) (Prod a_2 b_2)
  = geq r_a a_1 a_2 ∧ geq r_b b_1 b_2
```

Figure 4.1: Type-indexed equality function in the LIGD library

mechanisms of this library are simple and easier to understand than those of other more advanced libraries. The original LIGD paper (Cheney and Hinze 2002) gives a more detailed explanation of this approach.

In polymorphic lambda calculus it is impossible to write one parametrically polymorphic equality function that works on all datatypes (Wadler 1989). That is why the definition of equality in Haskell uses type classes, and ML uses equality types. The Eq type class provides the equality operator ==, which is overloaded for a family of types. To add a newly defined datatype to this family, a programmer defines an instance of equality for it. Thus, a programmer manually writes definitions of equality for every new datatype that is defined. For equality, type class deriving automates this process. However, this mechanism can only be used with a small number of type classes because it is hardwired into the language, making it impossible to extend or change by the programmer.

With generic programming, we can define equality once and use it on a large family of datatypes. Such functions are called generic functions. The introduction of a new datatype does not require redefinition or extension of an existing generic function. We merely need to describe the new datatype to the library, and all existing and future generic functions will be able to handle it.

Below we give a brief introduction to generic programming and the terminology that we use throughout this chapter.

A *type-indexed function* (TIF) is a function that is defined on every type of a family of types. We say that the types in this family index the TIF, and we call the type family a universe. A TIF is defined by case analysis on types: each type is assigned a function that acts on values of that type. As a familiar example, consider the TIF equality implemented using Haskell type classes. The universe consists of the types that are instances of the Eq type class. Equality is given by the == method of the corresponding instance. And the case analysis on types is provided by instance selection.

Haskell type classes are only one of the possible implementations of TIFs. In this section we use LIGD with Generalized Algebraic Datatypes (GADTs) (Xi et al. 2003; Cheney and Hinze 2003; Peyton Jones et al. 2006) to implement TIFs. We start with the equality TIF which is indexed by a universe consisting of units, sums and products. Figures 4.1 and 4.2 show the definitions. Note that in Haskell type variables appear-

```
data Unit      = Unit
data Sum a b   = Inl a | Inr b
data Prod a b  = Prod a b
```

Figure 4.2: Unit, sum and product datatypes

ing in type signatures are implicitly universally quantified. The first argument of the function is a *type representation*, which describes the type of the values that are to be compared (second and third arguments). Haskell does not allow functions to depend on types, so here types are represented by a GADT. This has the advantage that case analysis on types can be implemented by pattern matching, a familiar construct to functional programmers. The GADT represents the types of the universe consisting of units, sums and products:

```
data Rep t where
  RUnit  :: Rep Unit
  RSum   :: Rep a → Rep b → Rep (Sum a b)
  RProd  :: Rep a → Rep b → Rep (Prod a b)
```

geq has three *type-indexed function cases*, one for each of the base types of the universe.

TIF *instantiation* is the process by which we make a TIF specific to some type t, so that we can apply the resulting function to t values. In LIGD the instantiation process is straightforward: *geq* performs a fold over Rep t using pattern matching, and builds an equality function that can be used on t values. In other approaches, instantiation uses, for example, the type class system.

Now we want to instantiate equality on lists. Since a generic function can only be instantiated on types in the universe, we extend our universe to lists. There are two ways to do this. The first is *non-generic extension*, we extend our case analysis on types so that lists are handled by equality. In LIGD, this translates into the following: extend Rep with an *RList* constructor that represents lists, and extend equality with a case for *RList*:

$$geq\ (RList\ r_{\mathsf{a}})\ xs\ ys = \ldots$$

The second way to implement extension is *generic extension*: we describe the structure of the list datatype in terms of types inside the universe. The consequence is that instantiation to lists does not need a special case for lists, but reuses the existing cases for sums, products and units. To make the idea more concrete let us have a look at how type structure is represented in LIGD.

In LIGD, the structure of a datatype b is represented by the following Rep constructor.

```
RType :: Rep c → EP b c → Rep b
```

The type c is the *structure representation* type of b, where b can be embedded in c. The embedding is witnessed by a pair of embedding and projecting functions translating between b and c values.

$$\textbf{data}\ \mathsf{EP}\ \mathsf{b}\ \mathsf{c} = EP\{from :: (\mathsf{b} \to \mathsf{c}), to :: (\mathsf{c} \to \mathsf{b})\}$$

In LIGD, constructors are represented by nested sum types and constructor arguments are represented by nested product types. The structure representation type for lists is Sum Unit (Prod a [a]), and the embedding and projection for lists are as follows:

$$\begin{aligned}
&fromList :: [\mathsf{a}] \to \mathsf{Sum\ Unit\ (Prod\ a\ [a])}\\
&fromList\ [\,] &&= Inl\ Unit\\
&fromList\ (a : as) &&= Inr\ (Prod\ a\ as)\\
&toList :: \mathsf{Sum\ Unit\ (Prod\ a\ [a])} \to [\mathsf{a}]\\
&toList\ (Inl\ Unit) &&= [\,]\\
&toList\ (Inr\ (Prod\ a\ as)) &&= a : as
\end{aligned}$$

To extend the universe to lists, we use *RType*:

$$\begin{aligned}
&rList \quad :: \mathsf{Rep\ a} \to \mathsf{Rep}\ [\mathsf{a}]\\
&rList\ r_{\mathsf{a}} = RType\ (RSum\ RUnit\ (RProd\ r_{\mathsf{a}}\ (rList\ r_{\mathsf{a}})))\ (EP\ fromList\ toList)
\end{aligned}$$

Generic equality is still missing a case to handle datatypes that are represented by *RType*. The definition of this case is given below. It takes two values, transforms them to their structure representations and recursively applies equality.

$$geq\ (RType\ r_{\mathsf{a}}\ ep)\ t_1\ t_2 = geq\ r_{\mathsf{a}}\ (from\ ep\ t_1)\ (from\ ep\ t_2)$$

In summary, there are two ways to extend a universe to a type T. Non-generic extension requires type-specific, ad-hoc cases for T in type-indexed functions, and generic-extension requires a structure representation of T but no additional function cases. This is a distinguishing feature between type-indexed functions and generic functions. The latter include a case for *RType*, which allows them to exploit the structure of a datatype to apply generic uniform behaviour to values of that datatype; while the former do not have a case for *RType*, and rely exclusively on non-generic extension.

In LIGD, sums, products, and units are used to represent the structure of a datatype. Other choices are possible. For example, PolyLib includes the datatype Fix in its universe, to represent the recursive structure of datatypes. We refer to these representation choices as *generic views* (Holdermans et al. 2006). Informally, a view consists of base (or view) types for the universe (for example Sum and Prod) and a convention to represent structure, for example, the fact that constructors are represented by nested sums. The choice of a view often has an impact on the expressiveness of a library, that is, which generic function definitions are supported and what are the set of datatypes on which generic extension is possible.

4.3 Design of the benchmark suite

Most previous work on datatype-generic programming focuses on either increasing the number of scenarios in which generic programming can be applied, or on obtaining the same number of scenarios using fewer or no programming language extensions. For example, Hinze's work on "Polytypic values possess polykinded types" (Hinze 2000c) shows how to define generic functions that work on types of arbitrary kinds, instead of on types of a particular kind, and "Generics for the Masses" (Hinze 2006) shows how to do a lot of generic programming without using Haskell extensions. Both goals are achieved by either inventing a new generic programming approach altogether, or by extending an existing approach. We have collected a number of typical generic programming scenarios from the literature. These are used as a guide to design our benchmark suite. The intuition is that the evaluation of a library should give an accurate idea of how well the library supports the generic programming scenarios. We list the scenarios below:

- Generic versions of Haskell type class functionality such as equality (Eq), comparison (Ord) and enumeration (Enum) (Jansson and Jeuring 1998b).
- Serialization and deserialization functions such as *read* and *show* in Haskell (Jansson and Jeuring 2002).
- Traversals to query and modify information in datatypes (Lämmel and Peyton Jones 2003).
- Functions like map, crush, and transpose, which manipulate elements of a parametrized datatype such as lists (Jansson and Jeuring 1998b; Norell and Jansson 2004).
- Data conversion (Jansson and Jeuring 2002; Atanassow and Jeuring 2004).
- Test data generation (Koopman et al. 2003; Lämmel and Peyton Jones 2005).

We have identified the features that are needed from a generic library to implement the scenarios above. These features are used as criteria to characterise generic programming from a user's point of view, where a user is a programmer who *writes* generic programs. There are also users who only *use* generic programs (such as people that use **deriving** in Haskell), but the set of features needed by the latter kind of users is a subset of that needed by the former. Generic programming scenarios are not the only source of criteria, we also use the following sources:

- new features introduced to existing approaches such as Hinze (2000c),
- Comparing approaches to generic programming in Haskell (Hinze et al. 2007),
- the Haskell generics wiki page: `http://www.haskell.org/haskellwiki/Applications_and_libraries/Generic_programming`,

- our own ideas, based on several years experience with different approaches to generic programming.

We test whether the criteria are fulfilled with a benchmark suite. Each function in the suite tests whether or not an approach satisfies a particular criterion. For example, generic map cannot be implemented if the library does not support "abstraction over type constructors". Hence, if a library cannot be used to implement a function, it means that it does not support the criterion that the function is testing. Each function in the benchmark suite is a simplified version of one of the above programming scenarios.

Before introducing the functions used in the benchmark suite, we describe the datatypes on which they are used, and the related structure representation machinery.

4.3.1 Datatypes

The datatype construct in Haskell combines many aspects: type abstraction and application, recursion, records, local polymorphism, etc. In this section we introduce a number of datatypes, that cover many of these aspects. A generic programming library that can apply generic functions to one of these datatypes is said to support the aspects that the datatype requires in its definition.

Aspects that we test for are: parametrized types (type constructors, using type abstraction and application), simple and nested recursion, higher-kinded datatypes (with a parameter of kind $\star \rightarrow \star$) and constructor name information (to implement generic show).

Aspects that we do not test in this chapter are higher-rank constructors (explicit $\forall$ in the datatype declaration), existential types, GADTs, and parsing related information, namely record label names, constructor fixity, and precedence. The first three aspects are not tested because they are hardly supported by any of the libraries that we evaluate[1]. The last aspect, parsing-related information, can be incorporated using the same mechanisms as for providing constructor names, and therefore we do not add datatypes that test for this aspect.

Following each datatype definition we must also provide the machinery that allows universe extension for the particular library we are using. In LIGD, we represent the structure of a datatype T following the conventions described in Section 4.2. Each datatype T is associated to a value *rT* that contains the structure representation of the datatype and a pair of functions that convert to a representation value and back. Figure 4.3 shows the representations used to describe the structure of a datatype, and Figure 4.4 shows the signatures of a number of structure representations. See Section 4.2 for an example on generic extension to lists.

In our comparison, we use two datatypes (Salary and WTree) to check whether a library supports non-generic extension. Since in LIGD non-generic extension to a datatype requires the addition of a representation constructor to Rep, we extend it with two constructors for the datatypes Salary and WTree in order to formulate the non-generic extension tests. In general, post-hoc addition of constructors to the Rep datatype is a

[1]The spine view is the only approach supporting GADTs and, partially, existential types.

```
data Rep t where
  RUnit   :: Rep Unit
  RSum    :: Rep a → Rep b → Rep (Sum a b)
  RProd   :: Rep a → Rep b → Rep (Prod a b)
  RType   :: Rep a → EP b a → Rep b
  RSalary :: Rep Salary
  RWTree  :: Rep a → Rep w → Rep (WTree a w)
```

Figure 4.3: Definition of Rep. The two last constructors are not part of the LIGD library.

```
rCompany :: Rep Company
rDept    :: Rep Dept
rBinTree :: Rep a → Rep (BinTree a)
rWTree   :: Rep a → Rep w → Rep (WTree a w)
rGRose   :: (∀a . Rep a → Rep (f a)) → Rep a → Rep (GRose f a)
```

Figure 4.4: Type signatures of some type representations.

suboptimal idea that will break existing code. Concretely, the definition of *geq* in this chapter is for the first four constructors (*RUnit*, *RSum*, *RProd*, *RType*) of Rep, thus any use of *geq* on *RSalary* or *RWTree* will fail. We return to this problem in Section 4.5 where we discuss support for ad-hoc definitions in LIGD.

The company datatype. The Company datatype (Lämmel and Peyton Jones 2003) represents the organizational structure of a company.

```
data Company = C [Dept]
data Dept    = D Name Manager [DUnit]
data DUnit   = PU Employee | DU Dept
data Employee = E Person Salary
data Person  = P Name Address
data Salary  = S Float
type Manager = Employee
type Name    = String
type Address = String
```

To define the representation of Company we must also define the representation of the supporting datatypes Dept, DUnit, etc.

```
rCompany = RType (rList rDept) (EP fromCompany toCompany)
rDept = ...
```

Because Salary is used with non-generic extension, the representation uses *RSalary* directly:

$$rSalary = RSalary$$

Binary trees. The recursive BinTree datatype abstracts over the type of its elements stored in the leaves.

$$\textbf{data}\ \mathsf{BinTree\ a} = \mathit{Leaf}\ \mathsf{a} \mid \mathit{Bin}\ (\mathsf{BinTree\ a})\ (\mathsf{BinTree\ a})$$

Like lists, the representation depends on the representation of a:

$$\begin{aligned} &\mathit{rBinTree} :: \mathsf{Rep\ a} \to \mathsf{Rep\ (BinTree\ a)} \\ &\mathit{rBinTree}\ r_{\mathsf{a}} = \textbf{let}\ r = \mathit{rBinTree}\ r_{\mathsf{a}} \\ &\qquad\qquad \textbf{in}\ \mathit{RType}\ (\mathit{RSum}\ r_{\mathsf{a}}\ (\mathit{RProd}\ r\ r))\ (\mathit{EP}\ \mathit{fromBinT}\ \mathit{toBinT}) \end{aligned}$$

We omit the structure representations of the remaining datatypes since they follow the same pattern as *rBinTree*.

Trees with weights. We adapt the type of binary trees such that we can assign a weight, whose type is abstracted, to a (sub)tree.

$$\begin{aligned} \textbf{data}\ \mathsf{WTree\ a\ w} = &\ \mathit{WLeaf}\ \mathsf{a} \\ \mid &\ \mathit{WBin}\ (\mathsf{WTree\ a\ w})\ (\mathsf{WTree\ a\ w}) \\ \mid &\ \mathit{WithWeight}\ (\mathsf{WTree\ a\ w})\ \mathsf{w} \end{aligned}$$

Some of the generic function tests treat weights differently from elements, even if their types are the same. In our comparison, WTree is used to test both generic and non-generic extension. Tests on generic extension use a structure representation like that of BinTree and non-generic tests use constructor *RWTree* as the representation of WTree.

Generalized rose trees. Rose trees are (non-empty) trees whose internal nodes have a list of children instead of just two.

$$\textbf{data}\ \mathsf{Rose\ a} = \mathit{Node}\ \mathsf{a}\ [\mathsf{Rose\ a}]$$

We can generalize Rose by abstracting from the list datatype:

$$\textbf{data}\ \mathsf{GRose\ f\ a} = \mathit{GNode}\ \mathsf{a}\ (\mathsf{f}\ (\mathsf{GRose\ f\ a}))$$

The interesting aspect that GRose tests is higher-kindedness: it takes a type constructor argument f of kind $\star \to \star$.

Perfect trees. The datatype Perfect is used to model perfect binary trees: binary trees that have exactly 2^n elements, where n is the depth of the binary tree.

```
data Perfect a = Zero a | Succ (Perfect (Fork a))
data Fork a    = Fork a a
```

The depth of a perfect binary tree is the Peano number represented by its constructors. The datatype Perfect is a so-called *nested datatype* (Bird and Meertens 1998), because the type argument changes from a to Fork a in the recursion.

Nested generalized rose trees. The NGRose datatype is a variation on GRose that combines nesting with higher-kinded arguments: at every recursive call f is passed composed with itself:

```
data NGRose f a = NGNode a (f (NGRose (Comp f f) a))
newtype Comp f g a = Comp (f (g a))
```

4.3.2 Functions

Inspired by the generic programming scenarios given at the beginning of this section, we describe a number of generic functions for our benchmark suite.

It is not necessary to include all functions arising from the generic programming scenarios. If two functions use the same set of features from a generic programming library, it follows that if one of them can be implemented, the other can be implemented too. For example, the test case generator, generic read, and generic enumeration functions rely on library support for writing producer functions. So, in this case, it is sufficient to test whether the library can define a producer function.

Generic variants of type class functionality: Equality

Generic equality (in LIGD) takes a type representation argument Rep a and produces the equality function for a-values.

```
geq :: Rep a → a → a → Bool
```

Two values are equal if and only if they have the same constructor and the arguments of the constructors are pairwise equal. LIGD encodes constructors as nested sum types, so two constructors are the same only if they have the same sum-constructor (*Inl* or *Inr*). Constructor arguments are encoded as nested products, hence product equality requires the equality of corresponding components, see Fig. 4.1. LIGD ignores constructor names — only positions of constructors in the "constructor list" and positions of arguments in the "argument list" are taken into account.

The generic version of the Ord method, *compare*, would have type Rep a → a → a → Ordering. Like equality it takes two arguments and consumes them. Approaches that can implement equality can also implement comparison if constructor information is available (see the corresponding criterion in Section 4.4).

Serialization and deserialization: Show

The show function takes a value of a datatype as input and returns its representation as a string. It can be viewed as the implementation of **deriving** *Show* in Haskell. Its type is as follows:

$$gshow :: \mathsf{Rep\ a} \to \mathsf{a} \to \mathsf{String}$$

The function *gshow* is used to test the ability of generic libraries to provide constructor names for arbitrary datatypes. For the sake of simplicity this function is not a full replacement of Haskell's *show*:

- The generic show function treats lists in the same way as other algebraic datatypes. (Note that in the examples that follow we use $\rightsquigarrow$ to indicate reductions of expressions.)

  $$gshow\ [1,2] \rightsquigarrow \texttt{"(:) 1 ((:) 2 [])"}$$

  Note, however, that *gshow* is extended in one of the tests to print lists using Haskell notation. This is a separate test that is called *gshowExt*.

- It also treats strings just as lists of characters:

  $$gshow\ \texttt{"GH"} \rightsquigarrow \texttt{"(:) 'G' ((:) 'H' [])"}$$

- Other features that are not supported are constructor fixity, precedence and record labels.

Querying and transformation traversals

A typical use of generic functions is to collect all occurrences of elements of a particular constant type in a datatype. For example, we might want to collect all Salary values that appear in a datatype:

$$selectSalary :: \mathsf{Rep\ a} \to \mathsf{a} \to [\mathsf{Salary}]$$

We can instantiate this function to Company:

$$selectSalary\ rCompany :: \mathsf{Company} \to [\mathsf{Salary}]$$

Collecting values is an instance of a more general pattern: querying traversals. The function above can be implemented using (1) a general function (which happens to be generic) that performs the traversal of a datatype, and (2) a specific case that actually collects the Salary values. Such an implementation of *selectSalary* requires two features from a generic programming library:

- A generic function can have an ad-hoc (non-uniform) definition for some type. For example, *salaryCase* returns a singleton list of its argument if applied to a Salary value. Otherwise it returns the empty list.

  $$\begin{aligned}&\mathit{salaryCase} :: \mathsf{Rep\ a} \to \mathsf{a} \to [\mathsf{Salary}]\\&\mathit{salaryCase}\ \mathit{RSalary}\ \mathit{sal} = [\mathit{sal}]\\&\mathit{salaryCase}\ \mathit{rep}\qquad\ \_\ \ = [\,]\end{aligned}$$

  The LIGD library does not support this feature, but we extended the Rep type in Fig. 4.3 to be able to show what it would look like.

- A generic function can take another generic function as argument. Consider for example (the LIGD version of) the *gmapQ* function from the first SYB paper,

  $$\begin{aligned}&\mathit{gmapQ} :: (\forall \mathsf{a}\,.\,\mathsf{Rep\ a} \to \mathsf{a} \to \mathsf{r}) \to \mathsf{Rep\ b} \to \mathsf{b} \to [\mathsf{r}]\\&\mathit{gmapQ}\ f\ \mathit{rT}\ (K\ a_1 \ldots a_n) \rightsquigarrow [f\ \mathit{rT}_1\ a_1, \ldots, f\ \mathit{rT}_n\ a_n]\end{aligned}$$

  This function takes three arguments: a generic function f, a type representation and a value of that type. If the value is a constructor K applied to a number of arguments, *gmapQ* returns a list of f applied to each of the arguments.

  Using *salaryCase* as argument it gives:

  $$\begin{aligned}&\mathit{gmapQ}\ \mathit{salaryCase}\ (\mathit{rList}\ \mathit{rSalary})\ (S\ 1.0 : [S\ 2.0])\\&\quad\rightsquigarrow [\mathit{salaryCase}\ \mathit{rSalary}\qquad\quad (S\ 1.0)\\&\qquad , \mathit{salaryCase}\ (\mathit{rList}\ \mathit{rSalary})\ [S\ 2.0]]\\&\quad\rightsquigarrow [[S\ 1.0], [\,]]\end{aligned}$$

It is not a good idea to test for both features with one single test case in our suite: if a library does not support one of them the other will remain untested. For this reason we test these two features separately, using the functions *selectSalary* and *gmapQ*:

$$\begin{aligned}&\mathit{selectSalary} :: \mathsf{Rep\ a} \to \mathsf{a} \to [\mathsf{Salary}]\\&\mathit{gmapQ} :: (\forall \mathsf{a}\,.\,\mathsf{Rep\ a} \to \mathsf{a} \to \mathsf{r}) \to \mathsf{Rep\ a} \to \mathsf{a} \to [\mathsf{r}]\end{aligned}$$

Transformation traversals. An obvious variation on queries are transformation traversals. A typical example of such a traversal consists of transforming some nodes while performing a bottom-up traversal. Function *updateSalary* increases all occurrences of Salary by some factor in a value of an arbitrary datatype.

$$\begin{aligned}&\mathit{updateSalary} :: \mathsf{Float} \to \mathsf{Rep\ a} \to \mathsf{a} \to \mathsf{a}\\&\mathit{updateSalary}\ 0.1\ (\mathit{rList}\ \mathit{rSalary})\ [S\ 1000.0, S\ 2000.0]\\&\quad\rightsquigarrow [S\ 1100.0, S\ 2200.0]\end{aligned}$$

Transformations on constructors. The *updateSalary* function traverses datatypes other than Salary generically, in other words the traversal is performed on the structure representation using the cases for products, sums and units. It follows that it is unnecessary to supply ad-hoc traversal cases for such datatypes.

The ad-hoc behaviour in *updateSalary* targets a particular datatype. Constructor cases (Clarke and Löh 2003), a refinement of this idea, introduce ad-hoc behaviour that instead targets a particular constructor. Suppose we want to apply an optimization rule $x + 0 \mapsto x$ to values of a datatype that consists of a large number of constructors. Ideally, we want a rewrite function that has an ad-hoc case for sums, and traverses other constructors generically.

The benchmark suite includes the function *rmWeights* to test ad-hoc behaviour for constructors. This function removes the weight constructors from a WTree:

```
rmWeights (RWTree RInt RInt)
          (WBin (WithWeight (WLeaf 42) 1)
                (WithWeight (WLeaf 88) 2))
  ⇝ (WBin (WLeaf 42) (WLeaf 88))
```

The definition of the transformation handles the *WithWeight* constructor and lets the remaining constructors be handled by the generic machinery.

```
rmWeights :: Rep a → a → a
rmWeights r@(RWTree ra rw) t =
  case t of
    WithWeight t' w → rmWeights r t'
    t'              → ... handle generically ...
... rest of definition omitted ...
```

The second arm of the case traverses the structure representation of t' generically rather than matching *WBin* and *WLeaf* explicitly. The full code of the function is shown in Fig. 4.5.

The last line of the definition uses *rWTree* to traverse the structure representation of t'. Because it is essential that remaining *WithWeight* constructors in t' are removed, the definition of *rWTree* has to be altered for this function. The recursive occurrences of WTree have to be represented by *RWTree* rather than *rWTree* as is usually done in other structure representations. In this way traversals of the subtrees will again be handled by the ad-hoc case (see Fig. 4.5).

Abstraction over type constructors: crush and map

The function *crushRight* (Meertens 1996) is a generic fold-like function. Typical instances are summing all integers in a list, or flattening a tree into a list of elements.

```
rmWeights :: Rep a → a → a
rmWeights RUnit          Unit           = Unit
rmWeights (RSum ra rb)   (Inl x)        = Inl (rmWeights ra x)
rmWeights (RSum ra rb)   (Inr x)        = Inr (rmWeights rb x)
rmWeights (RProd ra rb)  (Prod a b)     = Prod (rmWeights ra a) (rmWeights rb b)
rmWeights (RType ra ep)  t              = to ep (rmWeights ra (from ep t))
rmWeights (RWTree ra rw) (WithWeight t w) = rmWeights (RWTree ra rw) t
rmWeights (RWTree ra rw) t              = rmWeights (rWTree  ra rw) t

rWTree :: Rep a → Rep w → Rep (WTree a w)
rWTree ra rw = let r = RWTree ra rw
               in RType (RSum ra (RSum (RProd r r) (RProd r rw)))
                        (EP fromWTree toWTree)
```

Figure 4.5: Generically remove weights from a WTree.

$$sumList :: [\mathsf{Int}] \to \mathsf{Int}$$
$$sumList\ [2,3,5,7] \rightsquigarrow 17$$
$$flattenBinTree :: \mathsf{BinTree\ a} \to [\mathsf{a}]$$
$$flattenBinTree\ (Bin\ (Leaf\ 2)\ (Leaf\ 1)) \rightsquigarrow [2,1]$$

The generic version of these functions abstracts over the type of the structure:

$$crushRight :: \mathsf{Rep'\ f} \to (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{f\ a} \to \mathsf{b} \to \mathsf{b}$$

The function *crushRight* traverses the f a structure accumulating a value of type b, which is updated by combining it with every a-value that is encountered during the traversal.

So far, generic functions use a type representation that encodes types of kind $\star$. Lists are not an exception: $rList\ r_a$ represents fully applied list types. To define *crushRight* we switch to a type representation that encodes types of kind $\star \to \star$. This is why we use Rep′ instead of Rep (and below *rList*′ instead of *rList*). This is a common situation: to increase expressiveness of a generic library the type representation is adjusted. This is unfortunate because different type and structure representations are usually incompatible.

Functions *sumList* and *flattenBinTree* are obtained by instantiating *crushRight* on lists or trees with the appropriate arguments:

```
sumList         xs = crushRight rList' (+) xs 0
flattenBinTree bt = crushRight rTree' (:)  bt []
```

How are generic queries different from *crushRight*? We could for example define a function *selectInt* to flatten a BinTree Int into a list of Int values. There are two differences. First, if the BinTree elements were booleans instead of integers, we would

need a different querying function: *selectBool*. With *flattenBinTree* we do not have this problem because it is parametrically polymorphic in the elements of the datatype.

The second difference is about the type signature of the querying function. Suppose now that we want to flatten WTree Int Int into a list of weights.

$$\begin{aligned}&\mathit{flattenWTWeights} :: \mathsf{WTree\ a\ w} \to [\mathsf{w}]\\&\mathit{flattenWTWeights}\ (\mathit{WBin}\ (\mathit{WithWeight}\ (\mathit{WLeaf}\ 1)\ 2)\ (\mathit{WithWeight}\ (\mathit{WLeaf}\ 3)\ 4))\\&\quad \rightsquigarrow [2,4]\end{aligned}$$

This is just an instance of *crushRight*:

$$\mathit{flattenWTWeights}\ \mathit{tree} = \mathit{crushRight}\ \mathit{rWTree}'\ (:)\ \mathit{tree}\ [\,]$$

where *rWTree*′ represents WTree a for any a. In contrast, for *selectInt*, there is no difference between Int-weights and Int-elements in the tree. So it gives the following incorrect result:

$$\begin{aligned}&\mathit{selectInt}\ (\mathit{WBin}\ (\mathit{WithWeight}\ (\mathit{WLeaf}\ 1)\ 2)\ (\mathit{WithWeight}\ (\mathit{WLeaf}\ 3)\ 4))\\&\quad \rightsquigarrow [1,2,3,4]\end{aligned}$$

Alternatively, we could use ad-hoc cases to solve this problem with queries. For example, we could wrap the w-elements in a **newtype**-type and give an ad-hoc case for it. We could also give an ad-hoc case for WTree where the second argument of *WithWeight* is extracted. This highlights the difference with *crushRight*: *crushRight* does not need ad-hoc cases, while traversal queries do.

To sum up, the difference with queries is that *crushRight* views the datatype as an application of a type constructor f to an element type a, and processes only a-values. In contrast, traversal queries do not make such element discrimination based on the type structure of the datatype.

Map. Generic map is to transformation traversals what *crushRight* is to query traversals. The *gmap* function takes a function and a structure of elements, and applies the function argument to all elements in that structure. The type signature of *gmap* uses the same representation as *crushRight*:

$$\mathit{gmap} :: \mathsf{Rep'\ f} \to (\mathsf{a} \to \mathsf{b}) \to \mathsf{f\ a} \to \mathsf{f\ b}$$

The best known instance is the map function on lists, but we also have instances like

$$\mathit{gmap}\ \mathit{rBinTree}' :: (\mathsf{a} \to \mathsf{b}) \to \mathsf{BinTree\ a} \to \mathsf{BinTree\ b}$$

In general, *gmap* can be viewed as the implementation of **deriving** for the Functor type class in Haskell.

Another function, generic transpose, is representative not only of abstraction over type constructors but also of data conversion functions. We discuss it next.

Data conversion: transpose

A data conversion function has type $\mathsf{T} \to \mathsf{T}'$: it converts T values into T' values. Jansson and Jeuring (2002) and Atanassow and Jeuring (2004) discuss generic approaches to build conversion functions. It turns out that there is no need to include the conversion functions from these sources, because the conversion functions are built out of simpler generic functions which are already accounted for in our scenarios. The former paper uses a combination of serialization, deserialization, and abstraction over type constructors. The latter paper composes serialization and deserialization functions that exploit isomorphisms in the intermediate structures.

A more sophisticated version of data conversion is the generic transpose function, described in Norell and Jansson (2004). Although this function can be implemented using a combination of serialization, deserialization, and abstraction over type constructors, it is an interesting challenge for library approaches because it abstracts over two type constructors.

The generic transpose function is a generalization of the function *transpose* that is defined in the Haskell standard library. Since this function abstracts over two type constructors it takes two type representations:

$$gtranspose :: \mathsf{Rep}'\ \mathsf{f} \to \mathsf{Rep}'\ \mathsf{g} \to \mathsf{f}\ (\mathsf{g}\ \mathsf{a}) \to \mathsf{g}\ (\mathsf{f}\ \mathsf{a})$$

Instantiating both f and g to the list type we obtain *transpose* again.

$$transpose = gtranspose\ rList'\ rList'$$
$$transpose\ [[1,2,3],[4,5,6]] \rightsquigarrow [[1,4],[2,5],[3,6]]$$

It can also be instantiated to other datatypes:

$$gtranspose\ rList'\ rBinTree'\ [Leaf\ 1, Leaf\ 2, Leaf\ 3]$$
$$\quad \rightsquigarrow Leaf\ [1,2,3]$$

Test data generation: Fulltree

Testing is a common methodology to detect program flaws. The cost of writing tests with good coverage can be reduced by using libraries such as QuickCheck (Claessen and Hughes 2000) in which random test values are generated automatically.

The user of QuickCheck is required to define a type class instance for every datatype for which value generation is desired. The definition of the type class instance is manual work that can be eliminated by means of generic programming. To be useful in this scenario, a generic programming library must support the generation of datatype values. This is tested by *gfulltree*, a simple value enumerator for datatypes.

The *gfulltree* function takes a representation of a container datatype as input and returns all possible values of the represented datatype up to a given depth. Note that the depth argument only makes sense with a *recursive* datatype. Here is the type signature of *gfulltree* and some examples of its usage:

$$
\begin{array}{l}
\mathit{gfulltree} :: \mathsf{Rep\ a} \to \mathsf{Int} \to [\mathsf{a}] \\
\mathit{gfulltree}\ (\mathit{rList\ RUnit})\ 4 \rightsquigarrow [[\,],[\mathit{Unit}],[\mathit{Unit},\mathit{Unit}],[\mathit{Unit},\mathit{Unit},\mathit{Unit}]] \\
\mathit{gfulltree}\ (\mathit{rBinTree\ RUnit})\ 4 \rightsquigarrow \\
\quad [\mathit{Leaf\ Unit} \\
\quad ,\mathit{Bin}\ (\mathit{Leaf\ Unit}) \qquad\qquad\qquad\quad (\mathit{Leaf\ Unit}) \\
\quad ,\mathit{Bin}\ (\mathit{Leaf\ Unit}) \qquad\qquad\qquad\quad (\mathit{Bin}\ (\mathit{Leaf\ Unit})\ (\mathit{Leaf\ Unit})) \\
\quad ,\mathit{Bin}\ (\mathit{Bin}\ (\mathit{Leaf\ Unit})\ (\mathit{Leaf\ Unit}))\ (\mathit{Leaf\ Unit}) \\
\quad ,\mathit{Bin}\ (\mathit{Bin}\ (\mathit{Leaf\ Unit})\ (\mathit{Leaf\ Unit}))\ (\mathit{Bin}\ (\mathit{Leaf\ Unit})\ (\mathit{Leaf\ Unit}))]
\end{array}
$$

4.3.3 More general representations of type constructors

The generic functions above use type representations over types of kind $\star$:

$$\mathit{geq} :: \mathsf{Rep\ a} \to \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$$

and over types of kind $\star \to \star$

$$\mathit{gmap} :: \mathsf{Rep'\ f} \to (\mathsf{a} \to \mathsf{b}) \to \mathsf{f\ a} \to \mathsf{f\ b}$$

In order to apply *gmap* to a datatype such as WTree yet a third representation on types of kind $\star \to \star \to \star$ would have to be defined. It is not good practice to define a new representation every time we encounter a kind that was not previously representable. Having many representation types increases the burden for the user of generic programming because universe extension and generic functions must be reimplemented when new representations are added.

Libraries such as LIGD and EMGM support the definition of generic functions that abstract over type constructors of arbitrary kinds. Instead of using a set of representations indexed by kinds, these libraries use a set of representations indexed by the *arity* of a generic function. Consider the type signature of a generic function over type constructors such as generic map. The arity of *gmap* is the occurrence count of the abstracted type constructor in the signature. For instance, *gmap* has arity two and the generalization of *zipWith* to arbitrary type constructors would have arity three.

The advantage of an arity-based approach is that *gmap* must not be redefined when datatypes of different kinds are involved. However, such approaches must still have as many representations as there are arities in use, therefore the user still needs to implement generic universe extension repeatedly. We discuss these issues in Section 4.7.

The use of arities to abstract over type constructors of arbitrary kinds was introduced by Hinze (2000c). The first Haskell library to use arity-based representations was described by (Hinze 2006).

4.4 Criteria

This section describes the criteria used to evaluate the generic programming libraries. We have grouped criteria around three aspects:

Types
- Universe Size
- Subuniverses

Expressiveness
- First-class generic functions
- Abstraction over type constructors
- Separate compilation

Expressiveness (continued)
- Ad-hoc definitions for datatypes
- Ad-hoc definitions for constructors
- Extensibility
- Multiple arguments
- Multiple type representation arguments
- Constructor names
- Consumers, transformers, and producers

Usability
- Performance
- Portability
- Overhead of library use
- Practical aspects
- Ease of use and learning

Figure 4.6: Criteria overview

- Types: what are the datatypes to which generic functions can be applied?
- Expressiveness: what kind of generic programs can be written?
- Usability: is the library easy to use and install? Is it portable? What is the performance of the library?

Figure 4.6 summarizes the criteria and the organization. In this section we describe the evaluation criteria and, when possible, we illustrate them with code.

Types

- **Universe Size**. What are the types that a generic function can be used on? The more types a generic function can be used on, the bigger the universe size for that library. Different approaches implement generic universe extension in different ways, hence the sizes of their universes can differ.

  Ideally, we would like to know whether a given library supports generic extension to nested and higher-kinded datatypes. But the claim that universe extension applies to, for example, nested datatypes is impractical to verify. It would require a rigorous proof that covers all nested datatypes.

  Instead, we take a less ambitious alternative to estimate the size of the universe. We test whether a given approach supports extension to a number of datatypes, each of which demonstrates a particular datatype property. We test universe extension on lists, BinTree and WTree (regular datatypes), GRose (higher-kinded), Perfect (nested), NGRose (higher-kinded and nested), and Company (mutually recursive).

- **Subuniverses**. Is it possible to restrict the use of a generic function to a particular set of datatypes, or to a subset of all datatypes? Will the compiler flag uses on datatypes outside that subuniverse as a type error?

Expressiveness

- **First-class generic functions**. Can a generic function take a generic function as an argument? This is tested by *gmapQ*, the function that applies a generic function argument to all constructor arguments:

  $$\begin{aligned} &\mathit{gmapQ}\ (\mathit{rList}\ \mathit{RInt})\ \mathit{gshow}\ (1:[2]) \\ &\quad \rightsquigarrow [\mathit{gshow}\ \mathit{RInt}\ 1, \mathit{gshow}\ (\mathit{rList}\ \mathit{RInt})\ [2]] \\ &\quad \rightsquigarrow [\texttt{"1"},\texttt{"(:) 2 []"}] \end{aligned}$$

  Here *gshow* is applied to the two fields of the list constructor $(:)$, each having a different type, hence *gshow* must be instantiated to different types.

- **Abstraction over type constructors**. The equality function can usually be defined in an approach to generic programming, but a generalization of the map function on lists to arbitrary container types cannot be defined in all proposals. This criterion is tested by the *gmap* and *crushRight* generic functions.

- **Separate compilation**. Is generic universe extension modular? That is, can a datatype defined in one module be used with a generic function and type representation defined in other modules without the need to modify or recompile them? This criterion is tested by applying generic equality to BinTree, which is defined in a different module than equality and the library itself.

  ```
  module BinTreeEq where
  import LIGD    -- import LIGD representations
  import GEq     -- and geq
  data BinTree a = ...
  rBinTree ra = RType (...) (EP fromBinT toBinT)
  eqBinTree = geq (rBinTree RInt)
                  (Leaf 2) (Bin (Leaf 1) (Leaf 3))
  ```

- **Ad-hoc definitions for datatypes**. Can a generic function contain specific behaviour for a particular datatype, and let the remaining datatypes be handled generically? In this situation, ad-hoc, datatype-specific definitions are used instead of uniformly generic behaviour. This is tested by the *selectSalary* function, which consists of cases that perform a traversal over a datatype, accumulating the values collected by the Salary ad-hoc case (traversal code omitted for brevity):

  ```
  selectSalary :: Rep a → a → [Salary]
  selectSalary RSalary (S x) = [S x]
  ...
  ```

- **Ad-hoc definitions for constructors**. Can we give an ad-hoc definition for a particular constructor, and let the remaining constructors be handled generically? This is tested by the *rmWeights* function, which should have an explicit case to remove *WithWeight* constructors and the remaining constructors should be handled generically.
- **Extensibility**. Can the programmer non-generically extend the universe of a generic function in a different module? Because the extension meant here is non-generic, this criterion makes sense only if ad-hoc cases are possible. This criterion is tested by extending *gshow* with an ad-hoc case that prints lists using Haskell notation:

  ```
  module ExtendedGShow where
  import GShow    -- import definition of gshow
    -- ad-hoc extension
  gshow (RList r_a) xs = ...
  ```

- **Multiple arguments**. Consumer functions such as *gshow* and *selectSalary* have one argument that is generic. Can the approach define a function that consumes more than one generic argument, such as the generic equality function?
- **Multiple type representation arguments**. Can a function be generic in more than one type? That is, can a generic function, such as the generic transpose function, receive two or more type representations? The evaluation of this criterion is work in progress, so we do not include it in this chapter.
- **Constructor names**. Can the approach provide the names of the constructors to which a generic function is applied? This is tested by the *gshow* generic function.
- **Consumers, transformers, and producers**. Is the approach capable of defining generic functions that are:
  - consumers ($a \to T$): *gshow* and *selectSalary*
  - transformers ($a \to a$ or $a \to b$): *updateSalary* and *gmap*
  - producers ($T \to a$): *gfulltree*

Usability

- **Performance**. Some proposals use many higher-order functions to implement generic functions, others use conversions between datatypes and structure types. We have compared running times for some of the test functions for the different libraries.
- **Portability**. Few proposals use only the Haskell98 standard for implementing generic functions, most use (sometimes unimplemented) extensions to Haskell98, such as recursive type synonyms, multi-parameter type classes with functional dependencies, GADTs, etc. A proposal that uses few or no extensions is easier to port across different Haskell compilers.

- **Overhead of library use**. How much additional programming effort is required from the programmer when using a generic programming library? We are interested in (1) support for automatic generation of structure representations, (2) number of structure representations needed per datatype, (3) the amount of work to instantiate a generic function, and (4) the amount of work to define a generic function.
- **Practical aspects**. Is there an implementation? Is it maintained? Is it documented?
- **Ease of learning and use**. Some generic programming libraries use implementation mechanisms that make their use or learning more difficult.

4.4.1 Coverage of testable criteria

Criteria can be divided into testable and non-testable groups. Testable criteria are the ones that can be tested by means of a generic function in the benchmark suite. Figure 4.7 shows the coverage of testable criteria. The rows represent testable criteria and the columns represent the means of testing them. The first group of columns stand for the testing functions introduced in Section 4.3. The criteria that a generic function tests are marked with ●.

The second group of columns stand for datatypes that test generic universe extension. These tests check whether *geq* can be instantiated and applied to values of those types.

Some testing functions unavoidably require support of two criteria from a library. For example, the generic extension test on the GRose datatype requires separate compilation and higher-kinded datatypes. This brings up the problem that lack of support for the first criterion will cause failure of the test, which, according to our procedure (described in Section 4.3), means failure for the second criterion too. As a result, despite the fact that the second criterion remains untested, the criterion will be assumed non-supported by the library. This test would fail on Spine because it does not support separate compilation, but from the failure of the test it can erroneously be concluded that higher-kinded datatypes are not supported by Spine.

For this reason we have tried to avoid requiring more than one criterion to implement a testing function, but this is not always possible. In such a situation we cheat a little: we ignore the issue of separate compilation and test it separately. This is shown in Figure 4.7. The criteria that are normally needed but are ignored for the particular test (because of the more than one criterion per test issue) are marked with ○.

4.4.2 Design choices

The criteria that we have seen so far are interesting from a user's point of view. They inform the user on what generic programs can and cannot be written using the libraries. However, it is also illustrative to see the design choices that have been taken by the designers of these libraries, because a particular design choice may improve or hinder

	geq	*selectSalary*	*gmapQ*	*updateSalary*	*rmWeights*	*crush*	*gmap*	*gshow*	*gshowExt*	*gtranspose*	*gfulltree*
Universe Size											
Regular datatypes	●		●		●	●	●			●	●
Higher-kinded datatypes											
Nested datatypes											
Nested & higher-kinded											
Mutually recursive		●		●				●	●		
First-class generic functions			●								
Abstraction over type constructors						●	●				
Separate compilation	●					○	○	○		○	
Ad-hoc definitions for datatypes		●		●	○				●		
Ad-hoc definitions for constructors					●						
Extensibility									●		
Multiple arguments	●										
Multiple type representation arguments								●	●		●
Constructor names								●	●		
Consumers	●	●	●			●		●	●		
Transformers				●	●		●			●	
Producers											●

	BinTree	GRose	Perfect	NGRose	Company
Universe Size					
Regular datatypes	●				
Higher-kinded datatypes		●		●	
Nested datatypes			●	●	
Nested & higher-kinded				●	
Mutually recursive					●
First-class generic functions					
Abstraction over type constructors					
Separate compilation	○	○	○	○	○
Ad-hoc definitions for datatypes					
Ad-hoc definitions for constructors					
Extensibility					
Multiple arguments	●	●	●	●	●
Multiple type representation arguments					
Constructor names					
Consumers	●	●	●	●	●
Transformers					
Producers					

● The criterion is tested by the example: the criterion is needed to implement test.
○ The criterion is normally needed by test, but it is circumvented to test other criteria.

Figure 4.7: Functions and datatypes set out against criteria.

the support of an expressiveness criterion. For example, the use of type classes is essential to libraries that support extensibility, but they can also make the use of generic functions as first-class values more difficult.

In this chapter we look at two design choices:

- **Implementation mechanisms**. How are types and their structure represented at runtime? Are these representations handled explicitly (as arguments that can be pattern matched) or implicitly (as type class contexts)? Are they abstract (higher order, including functions) or concrete (first order syntax for types).
- **Views**. What are the views that the generic library supports? Examples of views are the sum of products view, the fixed point view, and the spine view. A library typically includes a type representation per view.

A third possible design choice is whether generic functions are instantiated by compile time specialization or by interpretation of type representations at runtime. Here we do not include this design choice, because all evaluated libraries use interpretation. Approaches that encode type and structure representations as datatypes are clearly doing interpretation of the representation values. Type class based approaches also perform interpretation: dictionary values are used at runtime. Of course, some specialization may take place if the compiler performs inlining in the generic program.

Why is this design decision important? If an approach would implement instantiation by compile-time specialization, that approach would most likely not support higher-order generic functions. This is because higher-orderness requires specializing the generic function argument at *runtime*, as opposed to compiled time.

4.5 Evaluation summary

We have implemented the benchmark using each of the generic programming libraries. This section gives a summary of the results. Figure 4.8 presents the results in a table. The criteria that a generic programming library supports are marked with ●. The ones that are not supported are marked with ○. If a criterion is partially supported, or if it requires unusual programming effort, it is marked with ◐ (in the text we say it scores *sufficient*). A more detailed evaluation can be found in Section 4.7. That section also discusses the design choices behind each of the evaluated libraries.

Universe Size. The PolyLib library is limited to regular datatypes (with one parameter). In RepLib and MultiRec, datatypes with higher-kinded arguments (GRose and NGRose) are not supported. Approaches such as SYB, SYB3, Uniplate, and EMGM, which are based on type classes, have trouble supporting NGRose; the three first do not support it at all, while EMGM supports it but loses some functionality. Furthermore, the SYB3 library does not support Perfect and it has an additional complication: BinTree is supported only if the instance is manually written, but not with the generated instance; we return to this problem when evaluating the generation of representations.

LIGD and Spine have the advantage of a large universe size: they support all datatypes in this test. Smash also supports all datatypes, but there are datatypes that require unusual effort to allow generic extension: Perfect, NGRose, and Company.

Subuniverses. The PolyLib, EMGM, RepLib, and Smash libraries support subuniverses.

First-class generic functions. First class generic functions are not supported in PolyLib, Uniplate, and MultiRec. The remaining approaches support this criterion. However, EMGM and Smash only score sufficient. In EMGM, there is additional complexity when defining such functions, while Smash requires a new different structure representation.

Abstraction over type constructors. The LIGD, PolyLib, EMGM, RepLib, Spine, and Smash libraries support abstraction over type constructors. However, PolyLib, and Spine only support abstraction over type constructors of kind $\star \rightarrow \star$, so the support of these approaches for this criterion is only sufficient. The SYB3, Uniplate, and RepLib libraries do not support this criterion. Recent work (Kiselyov 2008) shows that SYB supports the definition of functions such as *gmap* and *crushRight*.

Separate compilation. The only evaluated approach that does not support this criterion is Spine: generic universe extension requires recompilation of the generic machinery (type representation) and all generic functions.

Ad-hoc definitions for datatypes. The approaches that do not support ad-hoc cases are LIGD, PolyLib, and Spine.

Ad-hoc definitions for constructors. Ad-hoc definitions for constructors are supported by all approaches. In LIGD, the structure representation has to be adapted for this criterion to work and hence it only scores sufficient.

Extensibility. This criterion is supported by SYB3, EMGM, RepLib, and Smash.

Multiple arguments. Multiple argument functions are supported by almost all approaches, however, in some approaches, such as SYB and SYB3, the definitions can be rather complex, and therefore they score sufficient. In Smash the definition is not complex, but it requires a separate structure representation. The only library that fails to support multiple arguments is Uniplate.

Constructor names. Constructor names are supported by all evaluated approaches except Uniplate.

Consumers, transformers, and producers. Almost all libraries support definitions of functions in the three categories. However, there are libraries that use different structure representations for consumers and producers such as SYB, SYB3, Spine, and Smash. Smash in addition uses a different structure representation for transformations. Uniplate does not support producer functions.

Performance. We have used some of the test functions for a performance benchmark comparing running times for larger inputs. The results are very sensitive to small code differences and compiler optimizations so firm conclusions are difficult to draw, but the best overall performance score is shared between PolyP, EMGM, RepLib, Smash, and Uniplate.

Portability. The three most portable approaches are LIGD, EMGM, and Uniplate. The first approach relies on existential types and the other two on multi-parameter type classes, both extensions are very likely to be included in the next Haskell standard. Furthermore, multi-parameter type classes in EMGM are used in a non-essential way: the functionality of EMGM would only be slightly affected in their absence. The other approaches rely on non-portable Haskell extensions.

Overhead of library use. The SYB, SYB3, EMGM, RepLib, Uniplate and MultiRec libraries are equipped with automatic generation of representations. However, automatic generation in RepLib fails for type synonyms. In SYB3, the generated `Data` instance for `BinTree` causes non-termination when used with generic equality. In MultiRec, automatic generation fails for polymorphic datatypes.

The number of structure representations is high for libraries such as LIGD, EMGM, and RepLib. The reason is that type constructor abstraction in these approaches requires one representation per generic function arity. The number of representations in Smash is even higher due to the amount of relatively specialized representations. More information can be found in Section 4.7. The SYB, SYB3, and Spine approaches have one representation for consumers and another for producers. In addition, Spine has a representation to abstract over type constructors. PolyLib and Uniplate have only one representation.

The instantiation of a generic function is easier (for the programmer) in libraries that support implicit type representations, such as PolyLib, SYB, SYB3, Uniplate, Smash, EMGM, and RepLib. However, the last two libraries require additional effort to enable instantiation. Therefore PolyLib, SYB, SYB3, Smash, and Uniplate are the libraries that require the least effort to instantiate a generic function.

The work required to define a generic function is higher, in the sense that more implementation machinery is required, in LIGD, SYB3, and RepLib.

Practical aspects. The SYB, EMGM, RepLib, Uniplate, and MultiRec libraries have well-maintained and documented distributions. PolyLib has an official distribution, but it is not maintained anymore. The SYB3 library has two distributions: the

	LIGD	PolyLib	SYB	SYB3	Spine	EMGM	RepLib	Smash	Uniplate	MultiRec
Universe Size										
Regular datatypes	●	●	●	●	●	●	●	●	●	●
Higher-kinded datatypes	●	○	●	●	●	●	○	●	●	○
Nested datatypes	●	○	●	○	●	●	●	◐	●	○
Nested & higher-kinded	●	○	○	○	●	◐	○	◐	○	○
Mutually recursive	●	○	●	●	●	●	●	◐	●	●
Subuniverses	○	●	○	○	○	●	●	●	○	○
First-class generic functions	●	○	●	●	●	◐	●	◐	○	○
Abstraction over type constructors	●	◐	◐	○	◐	●	●	●	○	○
Separate compilation	●	●	●	●	○	●	●	●	●	●
Ad-hoc definitions for datatypes	○	○	●	●	○	●	●	●	●	◐
Ad-hoc definitions for constructors	◐	●	●	●	●	●	●	●	●	●
Extensibility	○	○	○	●	○	●	●	●	○	○
Multiple arguments	●	●	◐	◐	●	●	●	◐	○	●
Constructor names	●	●	●	●	●	●	●	●	○	●
Consumers	●	●	●	●	●	●	●	●	●	●
Transformers	●	●	●	●	●	●	●	◐	●	●
Producers	●	●	◐	◐	◐	●	●	◐	○	●
Performance	◐	◐	○	○	◐	●	◐	●	●	
Portability	●	○	○	○	○	●	○	○	●	○
Overhead of library use										
Automatic generation of representations	○	○	●	◐	○	●	◐	○	●	◐
Number of structure representations	4	1	2	2	3	4	4	8	1	1
Work to instantiate a generic function	●	●	●	●	●	◐	◐	●	●	●
Work to define a generic function	◐	●	●	◐	●	●	◐	●	●	●
Practical aspects	○	◐	●	◐	○	●	●	○	●	●
Ease of learning and use	◐	●	○	○	●	◐	○	○	●	○

● Supported criterion
○ Unsupported criterion
◐ Partially supported criterion or unusual programming effort required

Figure 4.8: Evaluation of generic programming approaches

official one does not compile under some versions of GHC (6.6, 6.8.1, 6.8.2) and the Hackage distribution does not have a number of useful combinators. Smash has an online distribution, but its interface is not as structured as, for example, SYB3. The remaining approaches, LIGD and Spine, do not have a well-maintained distribution.

Ease of learning and use. It is hard to determine how easy it is to learn how to use a library. We approximate this criterion by looking at the mechanisms used in the implementation of the libraries. We consider an approach easier if its implementation mechanisms are relatively simple such as for PolyLib and Uniplate (type classes), and Spine (GADTs). An approach is relatively difficult if it uses sophisticated implementation mechanisms, for example rank-2 typed combinators and abstraction over type classes as in SYB3. Intermediate approaches use advanced mechanisms only occasionally. One such approach is EMGM, which uses arity-based representations. More information can be found in Section 4.7.

4.6 Overview of generic programming libraries

A generic programming library provides an interface to achieve generic programming behaviour and uses certain mechanisms to implement it. These interfaces and mechanisms correspond to the concepts that we introduced in Section 4.2. In particular we can usually identify mechanisms corresponding to type representations, structure representations and functions that act on them. So two libraries may implement generic behaviour in very different ways, by providing different ways to encode structure representations, for example.

Before proceeding with the detailed evaluation we introduce below each of the compared libraries and relate them to the concepts introduced in Section 4.2. We focus in particular on how they implement case analysis on types and structure representation of datatypes. Many generic programming libraries can be obtained from the Hackage package database (`http://hackage.haskell.org/packages/hackage.html`). The introduction for those libraries give the name and version of the package used in the evaluation.

4.6.1 Lightweight implementation of Generics and Dynamics

The Lightweight implementation of Generics and Dynamics (LIGD) library was introduced by Cheney and Hinze (2002). The presentation in the current chapter largely follows the original presentation of LIGD. The difference is that the original Rep is not a GADT but a normal datatype. This datatype encodes the GADT by including conversion functions in the datatype constructors and by the use of existential types.

The generic view that LIGD uses is the sum of products view. The original LIGD paper does not include a view to abstract over type constructors, but it is well known how to do so: Hinze and Löh (2007) present a variant of LIGD called dictionary-passing style that abstracts over type constructors.

In this library case analysis on types is performed by means of pattern matching. The type structure of datatypes is represented by the *RType* constructor.

4.6.2 PolyLib

The pre-processor-based language extension PolyP (Jansson and Jeuring 1997, 1998b) was later packaged up as a more lightweight library (Norell and Jansson 2004) and this library is what we compare in this chapter. The library is limited to regular datatypes (with one parameter) so the supported universe is relatively small. But the smaller universe makes it possible to express a wider range of generic functions — the library contains definitions of folds and unfolds, traversals and even functions generic in two type parameters such as $transpose :: \ldots \Rightarrow d\ (e\ a) \rightarrow e\ (d\ a)$.

The limited universe means that PolyLib is not suitable as a general generic library — it is included here as a "classic reference" and because of its expressiveness.

PolyLib uses a combination of the fixed-point view and the sum of products view. Each TIF is defined as a type (constructor) class with one instance for each universe building block (unit, sum, product, composition, function, const, parameter, and recursion).

4.6.3 Scrap your boilerplate

In the Scrap your boilerplate (SYB) library (Lämmel and Peyton Jones 2003, 2004) generic functions are not programmed by pattern matching on the structural representation of a value, but rather by means of combinators. There are combinators for doing case analysis on types and for inspecting the structure of values.

Case analysis combinators exist in several variants: query combinators, transformation combinators, and monadic transformation combinators amongst others. Let us consider the combinators for queries: *mkQ* and *extQ*.

$$mkQ :: (\mathsf{Typeable\ a}, \mathsf{Typeable\ b}) \Rightarrow \mathsf{r} \rightarrow (\mathsf{b} \rightarrow \mathsf{r}) \rightarrow \mathsf{a} \rightarrow \mathsf{r}$$
$$extQ :: (\mathsf{Typeable\ a}, \mathsf{Typeable\ b}) \Rightarrow (\mathsf{a} \rightarrow \mathsf{r}) \rightarrow (\mathsf{b} \rightarrow \mathsf{r}) \rightarrow (\mathsf{a} \rightarrow \mathsf{r})$$

The *mkQ* combinator takes an ad-hoc case specific to b values and creates a polymorphic function that can be applied to any value a that is an instance of Typeable. If it is applied to a b value –that is, if at runtime it is determined that a and b are the same type– the function with type $\mathsf{b} \rightarrow \mathsf{r}$ is applied to it, otherwise the argument of type r is returned. The *extQ* combinator extends a polymorphic query built with *mkQ* with yet another ad-hoc case.

These combinators are implemented by means of type-safe casting, which ultimately relies on the function *unsafeCoerce*, an unsafe Haskell extension that converts a value from one type to any other type. They also rely on the Typeable type class, which provides runtime representations of types, so that the equality of two types can be established at runtime.

The structure of datatypes is represented by the higher-order combinators *gfoldl* and *gunfold*. These are used to write consumer and producer functions respectively. Datatypes whose structure is represented are instances of the Data type class.

```
class Typeable a ⇒ Data a where
  gfoldl :: Data a ⇒ (∀a b . Data b ⇒ w (a → b) → a → w b)
                   → (∀a . a → w a)
                   → a → w a
```

The instance for lists below makes *gfoldl* clearer.

```
instance Data a ⇒ Data [a] where
  gfoldl k z []     = z []
  gfoldl k z (x:xs) = (z (:) `k` x) `k` xs
```

The *gfoldl* function applies z to the constructor and applies the result to the arguments of the constructor using k. In essence, *gfoldl* exposes the constructor and the arguments to the k and z functions. Further explanations can be found in the original SYB paper and in the description of the Spine approach below.

The SYB library provides two structure representations of data, one through *gfoldl* for consumer functions, and another through *gunfold* for producer functions. Because of their types these functions need support for rank-2 polymorphism.

As an example, this is how *selectSalary* is implemented in SYB:

```
selectSalary :: Data a ⇒ a → [Salary]
selectSalary x = ([] `mkQ` salaryCase) x : concat (gmapQ selectSalary x))
  where salaryCase (sal :: Salary) = [sal]
```

The *mkQ* expression performs case analysis on types, x is added to the result if it is a Salary but not otherwise. Then *gmapQ* applies *selectSalary* to the children of x.

```
gmapQ :: Data a ⇒ (∀a . Data a ⇒ a → u) → a → [u]
```

4.6.4 Scrap your boilerplate, extensible with type classes

A serious drawback of Scrap your boilerplate is that it is not extensible: once a function (such as *selectSalary*) is defined, it cannot be extended with an ad-hoc case. This problem is solved by the extensible variant of Scrap your boilerplate (SYB3) (Lämmel and Peyton Jones 2005). The extended approach is still combinator based, indeed generic functions are written using combinators such as *gfoldl* and *gmapQ*.

```
class (Typeable a, Sat (ctx a)) ⇒ Data ctx a where
  gfoldl  :: Proxy ctx
          → (∀b c . Data ctx b ⇒ w (b → c) → b → w c)
          → (∀g . g → w g)
          → a → w a
  gmapQ :: Proxy ctx → (∀a . Data ctx a ⇒ a → r) → a → [r]
```

There is an additional type argument ctx to Data. This is the essential ingredient that allows extension. Both functions also take an additional Proxy argument, which is merely a way to inform the type checker what ctx type is used and the actual argument value is not important. We shall explain more about ctx shortly.

However, case analysis on types is no longer based on combinators such as *mkQ* and *extQ*. Instead, generic functions are defined as a type class and ad-hoc cases are given as instances. Consider *selectSalary*, for example:

```
class SelectSalary a where
  selectSalary :: a → [Salary]
```

The generic function is the method of a type class. It follows that case analysis on types is performed by the type class system. We show below how to write the generic and the Salary-specific cases:

```
  -- case for Salary
instance SelectSalary Salary where
  selectSalary sal = [sal]
  -- generic case, not complete yet
instance Data SelectSalary a ⇒ SelectSalary a where
  selectSalary x = gmapQ someProxy (...) x
```

The idea of the generic case is that we want to apply *selectSalary* recursively to the top-level sub-trees in *x*. Since *gmapQ* abstracts over the generic function that is applied, it follows that it has to abstract over the type class as well. However, Haskell does not support abstraction over type classes, so in this approach abstraction over type classes is emulated by means of dictionaries.

The first step is to define a dictionary datatype that represents SelectSalary instances.

```
data SelectSalaryD a = SelectSalaryD{selectSalaryD :: a → [Salary]}
```

Next, every generic function definition must include a Sat instance declaration. The Sat type class is used to enable SYB3 combinators to construct dictionaries of generic functions.

```
class Sat a where {dict :: a}
instance SelectSalary a ⇒ Sat (SelectSalaryD a) where
  dict = SelectSalaryD selectSalary
```

Finally, combinators that take generic functions as arguments, such as *gmapQ*, include Sat in their context to abstract over the dictionary argument. This becomes clearer in the definition of *gmapQ* for lists:

```
instance (Sat (ctx [a]), Data ctx a) ⇒ Data ctx [a] where
  gmapQ _f []     = []
  gmapQ _f (x:xs) = [f x, f xs]
```

The generic case of *selectSalary* looks as follows:

```
instance Data SelectSalaryD a ⇒ SelectSalary a where
  selectSalary x = concat (gmapQ selectSalaryProxy (selectSalaryD dict) x)
selectSalaryProxy :: Proxy SelectSalaryD
selectSalaryProxy = undefined
```

This approach uses the same structure representation as SYB. Case analysis on types is implemented using the type class system, and hence it no longer uses type safe casts. However, casts are still used in the library, for example in the definition of the generic equality function.

For the comparison we use the package `syb-with-class` version 0.4 which is available from the Hackage package database.

4.6.5 Scrap your boilerplate, spine view variant

The Scrap your Boilerplate variant introduced in Hinze et al. (2006) replaces the combinator based approach of SYB by a tangible representation of the structure of values, which is embodied by the Spine datatype:

```
data Spine :: ⋆ → ⋆ where
  Con :: a → Spine a
  (:$) :: Spine (a → b) → Typed a → Spine b
```

where the Typed representation is given by:

```
data Typed a = (:>){typeOf :: Type a, val :: a}
data Type :: ⋆ → ⋆ where
  IntR  :: Type Int
  ListR :: Type a → Type [a]
    ...
```

This approach represents the structure of datatype values by making the application of a constructor to its arguments explicit. For example, the list $[1,2]$ can be represented by *Con* (:) :$ (*IntR* :> 1) :$ (*ListR IntR* :> [2]).

Unlike in LIGD, there is no general purpose constructor like *RType* to support generic universe extension. Generic universe extension is achieved as follows: (1) the datatype must have a Type constructor that represents it, e.g. the *ListR* constructor for lists, and (2) the function *toSpine* that transforms a value to its structure representation must be extended to cover that type.

```
toSpine :: Type a → a → Spine a
toSpine (ListR t) []     = Con []
toSpine (ListR t) (x:xs) = Con (:) :$ (t :> x) :$ (ListR t :> xs)
...
```

In Spine, case analysis on types is done as in LIGD, by pattern matching on Type values.

Generic and non-generic universe extension in Spine require recompilation of type representations and generic functions. For this reason Spine cannot be used as a library, and so it is a design pattern rather than a library. The authors of Spine also describe an extensible variant of Spine that is based on type classes (and therefore can be used as a library), but we do not evaluate it in this paper. This variant uses techniques similar to those in SYB3, so we expect that both libraries have similar expressiveness.

Producer generic functions cannot be defined using Spine. To solve this deficiency the authors introduced a "type spine view" in Hinze and Löh (2006). In the evaluation we refer to both approaches as Spine. Both views, spine and the type spine view, correspond to *gfoldl* and *gunfold* in SYB. As the authors of Spine note, *gfoldl* is a fold over Spine values.

4.6.6 Extensible and modular Generics for the masses

The EMGM library (Hinze 2006; Oliveira et al. 2006) does not use a datatype like Rep to represent types. Instead the type representations are encoded in the type class Generic, where every represented type has a corresponding method:

```
class Generic g where
  unit  :: g Unit
  bool  :: g Bool
  plus  :: g a → g b → g (Sum a b)
  prod  :: g a → g b → g (Prod a b)
  view  :: EP b a → g a → g b
```

The type class abstracts over the signature of a generic function, here represented by g. To define a generic function, the programmer defines a type for the signature and then the definition is given in the instance declaration for that type. Consider, for example, the equality function. The signature type is defined as follows:

```
newtype Geq a = Geq { geq :: a → a → Bool }
```

The definition of the generic function resides in the Geq instance declaration.

```
instance Generic Geq where
  unit      = Geq (λUnit Unit → True)
  bool      = Geq (λx y → eqBool x y)
  plus a b  = Geq (λx y → case (x,y) of
                            (Inl xl, Inl yl)  → geq a xl yl
                            (Inr xr, Inr yr)  → geq b xr yr
                            _                 → False)
  prod a b  = Geq (λ(Prod a1 b1) (Prod a2 b2) → geq a a1 a2 ∧ geq b b1 b2)
  view ep a = Geq (λx y → geq a (from ep x) (from ep y))
```

The equality function is now defined for the universe of types comprising units, sums, products, and datatypes that have their structure represented by *view*, which is similar to the use of the *RType* constructor in LIGD. It follows that in EMGM case analysis on types is encoded using the methods of Generic. This is how the structure of lists is represented in EMGM.

```
rList :: Generic g ⇒ g a → g [a]
rList a = view listEP (unit `plus` (a `prod` rList a))
```

Extensibility of generic functions is achieved by means of defining sub-classes of Generic. For example, we define GenericList to enable ad-hoc definitions for lists:

```
class Generic g ⇒ GenericList g where
  list :: g a → g [a]
  list = rList
```

The default implementation of GenericList uses the structure representation for lists. Therefore we can request generic behaviour for list equality with an empty instance declaration:

```
instance GenericList Geq
```

But we can also give a definition of equality specific to lists:

```
instance GenericList Geq where
  list geqa = Geq (λx y → ...)
```

Now let us see how to apply generic equality:

```
geq (list bool) [True, False] [True, True] ⇝ False
```

It is possible to make the use of generic functions easier by making the type representations implicit. This is achieved by means of a type class:

```
class GRep g a where
  over :: g a
instance Generic g ⇒ GRep g Unit where
  over = unit
instance (Generic g, GRep g a, GRep g b) ⇒ GRep g (Prod a b) where
  over = prod over over
instance (GenericList g, GRep g a) ⇒ GRep g [a] where
  over = list over
```

Now, generic equality can be defined as follows:

```
gequal :: GRep Geq a ⇒ a → a → Bool
gequal x y = geq over
```

4.6.7 RepLib

The RepLib library (Weirich 2006) uses an ingenious combination of GADTs and type classes to implement generic functions. A generic function in this approach is implemented as a type class. Ad-hoc cases are given as an instance of this class. We use the *gsum* function from the original paper as an example.

```
instance GSum IntSet where
  gsum (IntSet xs) = gsum (nub xs)
```

Here we give an ad-hoc case for sets of integers. This case eliminates duplicate elements and calls generic sum on the resulting list.

What makes *gsum* a generic function, and not-merely type-indexed, is the default implementation, which exploits the structure of datatypes:

```
class Rep1 GSumD a ⇒ GSum a where
  gsum :: a → Int
  gsum = gsumR1 rep1
```

The structure representation for a is generated by *rep1*, a method of the Rep1 type class.

```
class Rep a ⇒ Rep1 c a where
  rep1 :: R1 c a
```

Now, *gsumR1* can use the representation produced by *rep1* to process its argument of type a.

But what happens if *gsumR1* needs to recursively apply *gsum* to a substructure inside a? In RepLib such recursive calls are allowed by parametrizing Rep1 over a dictionary type. In our example the dictionary is GSumD, which is defined as follows:

```
data GSumD a = GSumD{ gsumD :: a → Int }
```

The representation produced by *rep1* contains GSumD dictionaries. These dictionaries package sum instances for a values. These instances are used when sum is applied recursively to the argument of a constructor, for example. To produce such dictionaries, the programmer defining the generic function is required to define a Sat instance for GSumD, it suffices to say that Sat is used in Rep1 instances to produce dictionaries. We give the instance definition for GSumD below:

```
class Sat a where dict :: a
instance GSum a ⇒ Sat (GSumD a) where
  dict = GSumD gsum
```

Note that this instance uses GSum to produce dictionaries. This means that even GSum instances defined in other modules are used. This is what allows RepLib generic functions to be extensible. The technique of explicit dictionaries to abstract over type classes is the same as in “Scrap your boilerplate with class”.

The *gsumR1* function is the structure-based definition of generic sum.

```
gsumR1 :: R1 GSumD a → a → Int
gsumR1 Int1                x = ...
gsumR1 (Arrow r1 r2)       f = ...
gsumR1 (Data1 dt cons) x
  = case (findCon cons x) of
      Val emb rec kids → foldl_l (λca a b → (gsumD ca b) + a) 0 rec kids
gsumR1 _                   x = 0
```

The two first cases and the last one correspond to integers, functions and all other cases respectively. Without going into detail, the third case uses the structure representation of the datatype (stored in *Data1*) to (1) find the representation for the constructor at hand (using *findCon*), (2) convert the constructor arguments into a heterogenous list, and (3) fold over this list applying generic sum to the constructor arguments (using *foldl_l*). Note that the last step involves a recursive application of the function. Remember that the representation stores GSumD dictionaries for the constructor arguments. Suppose that *b* (a constructor argument) has type c and *ca* has type GSumD c, then we can apply the dictionary function using *gsumD*.

In this approach the structure of datatypes is represented by GADTs such as R1. Case analysis over types is performed by the type class system, because generic functions are implemented as type classes (for example, GSum).

The RepLib library has an alternative way to implement case analysis on types. It provides SYB-style combinators such as *mkQ* and *extQ*, these are implemented using type safe casts like in SYB. But these combinators use RepLib representations, rather than the Typeable class.

4.6.8 Smash your boilerplate

The 'Smash' approach is conceptually closely related to SYB. The latter uses a 'typecase' operation based on the run-time type representation (Typeable). The Smash approach uses a *compile-time* typecase operation. In both approaches, the structure of a new datatype is presented to the library (added to the universe) by declaring an instance of a special class: Data in SYB, LDat in Smash. A generic function is made of two parts. First, there is a term traversal strategy, identified by a label. One strategy may be to 'reduce' a term using a supplied reducing function (cf. fold over a tree). Another strategy may rebuild a term. The second component of a generic function is *spec*, the list of 'exceptions', or ad-hoc redefinitions. Each component of *spec* is a function that tells how to transform a term of a specific type. Exceptions override the generic traversal.

As an example, consider how *selectSalary* is defined in Smash:

```
selectSalary :: Company → [Salary]
selectSalary x = gapp (TL_red concat) (salaryCase :+: HNil) x
  where
  salaryCase :: Salary → [Salary]
  salaryCase s = [s]
```

Here the library function *gapp* is applied to *TL_red concat*, which selects a bottom-up traversal (parametrized with *concat*) on *x*. This traversal applies one of the ad-hoc cases (second argument of *gapp*) to the nodes of *x* being traversed. When traversing a node, the results of traversing the children are merged using the *concat* function. Note that ad-hoc cases are encoded as a heterogeneous list of functions. In the above example the list contains only one element.

This library implements case analysis using extensible record operations (Kiselyov et al. 2004), due to the way that ad-hoc cases are encoded. The structure representation is given once per datatype and per traversal strategy. To implement the functions in the test suite the following strategies are used:

- A rewriting strategy (*TL_recon*) that is used to implement functions such as *gmap* and *updateSalary*.
- A reduction strategy (*TL_red*) that is used to implement *selectSalary*.
- A reduction strategy that also provides access to constructor names (*TL_red_con*). This strategy is used to implement *gshow*.
- A twin traversal strategy (*TL_red_lockstep*) that is used to implement functions with multiple arguments such as equality.
- A shallow reduction traversal (*TL_red_shallow*) that is used to implement *gmapQ*.
- A couple of reduction traversals that abstract over $\star \to \star$-types (*TL_red_cr1*) and $\star \to \star \to \star$-types (*TL_red_cr2*).
- a traversal strategy for producer functions.

4.6.9 Uniplate

The Uniplate library provides a form of generic programming based on traversal combinators. There are two sorts of traversals: single type and multi-type traversals. Unlike SYB, Uniplate combinators do not require a type system that supports rank-2 types. This is because traversals are customized by functions that are monomorphic rather than polymorphic, as in SYB. An example of a Uniplate combinator is the bottom-up transformation traversal (*transform*).

```
transform :: Uniplate a ⇒ (a → a) → a → a
```

The *transform* traversal applies the function argument to every a-value that is contained within the a argument. The Uniplate type class is the analog of Data in SYB, it provides

a set of common traversals on top of which more sophisticated traversals are defined. The Uniplate type class is equipped with 10 traversal methods. However, all of them can be defined in terms of the fundamental *uniplate* operation (the analog in SYB is *gfoldl*):

$$uniplate :: \mathsf{Uniplate\ a} \Rightarrow \mathsf{a} \rightarrow ([\mathsf{a}], [\mathsf{a}] \rightarrow \mathsf{a})$$

This function takes an argument of type a and returns a pair of (1) a list of maximal substructures with type a, and (2) a function that rebuilds the argument using new values for those substructures. For example, *uniplate* (*Bin* (*Leaf* 1) (*Leaf* 2)) yields ([*Leaf* 1, *Leaf* 2], *Bin*). The *uniplate* function can be seen as the structure representation of a-values, because all combinator definitions ultimately rely on it.

In Uniplate, the arguments of traversals cannot perform case analysis on types, because they are monomorphic functions. Interestingly, this is not a limitation in practice, because in general the interesting case of a traversal is restricted to one type. This is also the case of functions *selectSalary* and *updateSalary* in our suite.

Uniplate also provides multi-type traversals using multi-parameter type classes. Consider the multi-type variant of *transform*:

$$transformBi :: \mathsf{Biplate\ b\ a} \Rightarrow (\mathsf{a} \rightarrow \mathsf{a}) \rightarrow \mathsf{b} \rightarrow \mathsf{b}$$

This combinator applies the function argument to all a-values that are contained in the b argument. For example, this is how Uniplate implements *updateSalary*:

$$increase :: \mathsf{Float} \rightarrow \mathsf{Company} \rightarrow \mathsf{Company}$$
$$increase\ k = transformBi\ (incS\ k)$$
$$incS :: \mathsf{Float} \rightarrow \mathsf{Salary} \rightarrow \mathsf{Salary}$$
$$incS\ k\ (S\ s) = S\ (s \star (1 + k))$$

Multi-type traversals are more flexible than single-type traversals, in that they allow the specification of an ad-hoc case on a type while doing the traversal on another.

4.6.10 MultiRec

The MultiRec library has been created to address the limitations of a fixed point view on datatypes while retaining the advantages of this view.

Case analysis on types is implemented by pattern matching a type representation value. Type representations are defined using a GADT. The structure of a datatype is encoded by a finite number of type constructors similar to those of PolyLib. More detailed information on this approach can be found in Chapter 3.

In the comparison, we use the package `multirec` version 0.2, available from the Hackage package database.

4.7 Evaluation

We now present the detailed evaluation. The results are summarized in Fig. 4.8.

Universe Size. What are the types that a generic function can be used on? That is, what are the datatypes to which generic universe extension is possible? This question is answered separately for each of the sub-criteria of universe size.

A library scores good on regular, higher-kinded datatypes, nested datatypes, higher-kinded and nested datatypes, and mutually recursive datatypes, if it can generically extend the universe to BinTree, GRose, Perfect, NGRose, and Company respectively, and apply generic equality to them. The library scores bad otherwise.

The LIGD approach can represent the structure of all datatypes in the universe size test, therefore it scores good on this criterion. The structure of a datatype T of kind $\star$ is represented as a Rep T value constructed with *RType*. For instance, the datatype Company is represented by *rCompany*, which has type Rep Company. Type constructors are represented by functions on representations. For instance, lists are represented by *rList* which has type Rep a → Rep [a]. The encoding can be generalized to higher-kinded parameters and nested datatypes: they are represented by higher-order functions with rank-2 types.

The PolyLib library is limited to regular datatypes (with one parameter) and cannot handle mutually recursive datatypes, so the set of datatypes (the universe) supported is relatively small.

The SYB library scores well on the universe size criteria, even for the Perfect datatype, which is nested. The GRose datatype presents difficulties because it has an type argument of kind $\star \to \star$, and so instances for Data and Typeable cannot be automatically derived. However, these instances can be written by the programmer, therefore generic universe extension to GRose is supported. Note that in the instance for GRose

```
instance (Typeable1 f, Typeable a, Data a, Data (f (GRose f a)))
  ⇒ Data (GRose f a) where ...
```

the instance head (GRose f a) reappears in the context. This implies that cycle-aware constraint resolution (Lämmel and Peyton Jones 2005) is required to type this program. In contrast, SYB does not support generic extension to NGRose. The reason is that in the NGRose instance

```
instance (Typeable1 f, Typeable a, Data a, Data (f (NGRose (Comp f f) a)))
  ⇒ Data (NGRose f a) where ...
```

the head (NGRose f a) becomes bigger in the context, namely f becomes Comp f f, and therefore cycle-aware resolution is not enough to type check programs using this instance.

The universe of SYB3 is even smaller than that of SYB: Perfect is not supported. The instance that is automatically derived looks as follows:

```
instance (Data ctx (Perfect (Fork a))), ...) ⇒ Data ctx (Perfect a) where
```

This instance has the same problems as the NGRose instance in SYB. Programs that use it, will not type check. The NGRose datatype is not supported for the same reason.

Universe extension for the BinTree datatype is supported, but we surprisingly have to manually write a Data instance for it. The reason is that *Derive*, the module that automatically generates representations, produces an erroneous Data instance. Indeed, the generated instance causes non-termination at runtime. The reason, we believe, is an erroneous Typeable dictionary at runtime, which causes looping when it is used to cast inside generic equality. We give BinTree a good score anyway, because this is a problem of *Derive*, rather than of support for regular datatypes.

The Spine approach has the advantage of a large universe size: it can handle all datatypes in the universe size test.

In EMGM, the structure of a datatype T is represented by a value of type Generic g ⇒ g T, which is built using the *view* method. Like in LIGD, type constructors are encoded as functions over representations, for example the type constructor list is encoded as Generic g ⇒ g a → g [a]. Higher-kinded datatypes such as GRose and even NGRose can also be encoded in EMGM. But NGRose cannot be used with implicit representations, because the type class that implements them (GRep), would need an instance that raises the same issues as the Data instance above. In summary, generic extension to NGRose is supported but at the cost of reduced functionality. Therefore EMGM scores good for all criteria, except for nested and higher kinded datatypes where it scores sufficient.

The RepLib and MultiRec libraries cannot represent datatypes with higher-kinded arguments. It follows that these libraries satisfy the tests for regular and mutually recursive datatypes only.

The Smash library represents the structure of all datatypes in the universe size test. Nested datatypes such as Perfect and NGRose present problems similar to those in other type class based approaches. However, it is possible to represent them with some effort. The Company datatype caused looping during type checking. A workaround is possible, but at the moment we have not identified the exact cause of the problem. Because of the difficulties mentioned, Smash scores sufficient on the problematic subcriteria.

The Uniplate library scores well on universe size, it can handle all datatypes except nested generalized rose trees. To support higher-kinded datatypes, the same instances for Data and Typeable are used as for SYB.

Subuniverses. Is it possible to restrict the use of a generic function to a particular set of datatypes? An approach scores good if uses of the generic function on datatypes outside of the set are flagged as compile-time errors.

The RepLib and EMGM approaches score good on this criterion. In both approaches the set of types to which a generic function can be instantiated is controlled by instance declarations. For example, if generic equality is to be used on lists, the programmer is expected to write the following instance (or an instance containing an ad-hoc definition):

```
instance GenericList Geq
```

otherwise compilation fails with a type checking error when applying equality to lists.

PolyLib also supports subuniverses - a TIF is limited to the instances defined and this is compiler checked.

In Smash, it is possible to specify which datatypes are not to be handled by a generic function. Therefore, Smash supports subuniverses "by exclusion".

First-class generic functions. Can a generic function take a generic function as an argument? If *gmapQ* can be implemented in a library such that it can be passed a generic function (for example, *gshow*) as argument, the library scores good. If *gmapQ* can be written but at the price of additional complexity the library scores sufficient. Otherwise, if *gmapQ* cannot be implemented, the library scores bad.

In LIGD, SYB, and Spine, a generic function is a polymorphic Haskell function, so it is a first-class value in Haskell implementations that support rank-2 polymorphism. Consider for example *gmapQ* in LIGD:

$$gmapQ :: (\forall \mathsf{a} \,.\, \mathsf{Rep\ a} \to \mathsf{a} \to \mathsf{r}) \to \mathsf{Rep\ b} \to \mathsf{b} \to [\mathsf{r}]$$

Here the function argument is polymorphic, which allows *gmapQ* to instantiate it to different types. In short, in LIGD, SYB, and Spine the generic function argument is just a normal functional argument, albeit a polymorphic one. It follows that LIGD, SYB, and Spine score good on this criterion.

EMGM scores sufficient because although EMGM supports first-class generic functions, they are implemented in a rather complicated way. The reason is that the type class system needs to track calls to generic functions. So we are forced to go from a relatively simple (but wrong) signature for GMapQ:

$$\textbf{data}\ \mathsf{GMapQ\ a} = \mathit{GMapQ}\{\, \mathit{gmapQ} :: (\ldots \to \mathsf{r}) \to \mathsf{a} \to [\mathsf{r}]\,\}$$

to a type signature that allows to track calls to the generic function argument. The new signature below abstracts over a type g, the signature of the function argument, and *garg*, which is the generic function argument itself.

$$\begin{aligned}&\textbf{data}\ \mathsf{GMapQ\ g\ a} = \mathit{GMapQ}\{\\ &\quad \mathit{garg} \quad :: \mathsf{g\ a},\\ &\quad \mathit{gmapQ} :: (\forall \mathsf{a} \,.\, \mathsf{g\ a} \to \mathsf{a} \to \mathsf{r}) \to \mathsf{a} \to [\mathsf{r}]\,\}\end{aligned}$$

This makes the definition of *gmapQ*, significantly more complex than other functions, such as generic equality.

The test function *gmapQ* can be defined with no difficulties in SYB3 and RepLib, which therefore score good.

In Uniplate, traversal combinators are parametrized over monomorphic functions and not over other generic functions, as is the case in SYB. It is not evident how *gmapQ* would be implemented in Uniplate, hence it scores bad.

In Smash, *gmapQ* is implemented using the *TL_red_shallow* reduction strategy. However, having a new strategy altogether, in place of using an existing one, imposes the burden of one more structure representation for the user. Therefore this library only scores sufficient.

MultiRec does not support higher-order generic functions. Since generic functions are based on type classes, we need a form of abstraction over type classes to implement *gmapQ*. Probably, it is possible to use the abstraction technique introduced in SYB3 to enable support for higher-order generic functions in MultiRec.

Abstraction over type constructors. Is a generic library able to define the generic functions *gmap* and *crushRight*? If a library can define both functions which can then be instantiated to *mapBinTree* and *flattenWTree*, which have types:

$$\textit{mapBinTree} :: (\mathsf{a} \to \mathsf{b}) \to \mathsf{BinTree\ a} \to \mathsf{BinTree\ b}$$
$$\textit{flattenWTree} :: \mathsf{WTree\ a\ w} \to [\mathsf{a}]$$

then the approach scores good. If the library can only support the definition of one of the functions or none, it scores sufficient or bad, respectively.

The LIGD, EMGM, and RepLib libraries support the definitions of *crushRight* and *gmap* by means of arity-based type representations (Section 4.3.3), and their instantiations yield functions *mapBinTree* and *flattenWTree* as required. Hence these libraries score good.

PolyLib includes the definition of *gmap* and *crushRight*. However these functions can be instantiated only to regular datatypes with kind $\star \to \star$. Therefore *flattenWTree* cannot be obtained from *crushRight* because WTree has kind $\star \to \star \to \star$. Therefore PolyLib scores sufficient.

In Smash, the definition of *gmap* and *crushRight* are supported. Generic map is implemented by means of the rewriting traversal strategy *TL_recon*. This strategy supports ad-hoc cases that can change the type of elements, so *gmap* can be instantiated to *mapBinTree*. The definition of *crushRight* uses two special purpose reduction strategies, one for $\star \to \star$-types and the other for $\star \to \star \to \star$-types.

Recent work by (Reinke 2008) and (Kiselyov 2008) shows that SYB supports the definition of *gmap* and *crushRight*. However, SYB scores sufficient only because of complexity in the definitions. For example, the definition of *gmap* uses direct manipulation of type representations, runtime casts, and the *gunfold* combinator. Furthermore, the programmer must take additional steps to ensure the totality of *gmap*. Indeed, the type of the function is too flexible and it can cause runtime failure if instantiated with the wrong types. Hence, the programmer must provide a wrapper that suitably restricts *gmap*'s type.

The Uniplate, MultiRec and SYB3 libraries represent types with kind $\star$ but they do not represent type constructors. It follows that they are unable to support the definitions of *gmap* and *crushRight*.

Separate compilation. Is generic universe extension modular? Approaches that can instantiate generic equality to BinTree without modifying and recompiling the function definition and the type/structure representation score good.

The LIGD, EMGM, and RepLib libraries score good on this criterion. In these approaches, the structure definition of BinTree uses type representation constructors (or

methods of the Generic type class in the case of EMGM) but requires no modifications to the type representation or generic function definition.

Approaches in which universe extension to a datatype is implemented with a type class instance (or a type family instance) support separate compilation. Such approaches are SYB, Uniplate, SYB3, PolyLib, Smash and MultiRec.

The Spine library scores bad on this criterion. The reason is that universe extension requires that the datatype, in this case BinTree, is represented by a constructor in the GADT that encodes types. Because this datatype is defined in a separate module, recompilation is required.

Ad-hoc definitions for datatypes. Can a generic function contain specific behaviour for a particular datatype, and let the remaining datatypes be handled generically? Moreover, the use of ad-hoc cases should not require modification and recompilation of existing code (for instance the type representations). If the function *selectSalary* can be implemented by a library using an ad-hoc case for the Salary datatype, it scores good. Otherwise, it scores bad.

In LIGD and Spine, the ad-hoc case in the *selectSalary* function would have to be given by pattern matching on the type representation constructor that encodes Salary, namely *RSalary*. However, this requires the type representation datatype to be extended, and hence the recompilation of the module that contains it and those containing completely unrelated generic functions. For this reason both approaches score bad.

PolyLib does not support ad-hoc cases. Ad-hoc cases are only useful for generic functions that encounter several different types in a traversal. In PolyLib, a datatype is modelled such that a generic function does not traverse sub-values that have a different type than that of the root. It follows that special behaviour for such differently typed values cannot be specified (because these values are not traversed) and therefore PolyLib scores bad on this criterion.

In SYB, ad-hoc cases for queries are supported by means of the *mkQ* and *extQ* combinators. Such combinators are also available for other traversals, for example transformations. The only requirement for ad-hoc cases is that the type being case-analysed should be an instance of the Typeable type class. The new instance does not require recompilation of other modules, so SYB scores good on this criterion.

The SYB3, EMGM, and RepLib libraries score good on this criterion. In SYB3 and RepLib, ad-hoc cases are given as an instance of the type class that implements the generic function. In EMGM, ad-hoc cases are given as an instance of Generic (or a subclass corresponding to the represented datatype). Because ad-hoc cases are given as type class instances, recompilation is not needed.

In Uniplate, it is possible to define datatype specific behaviour for a multi-type traversal. This is usually achieved by using a traversal combinator that is parametrized over (1) the type on which the traversal is performed, (2) the type for which the ad-hoc case is given, and (3) the ad-hoc case function. Function *transformBi* is such a combinator.

In Smash, a monomorphic ad-hoc definition is given as an element in the list of ad-hoc cases (a function of type ad_hoc_type → String in case of *gshow*). Smash performs

case analysis on types using a type equality operation implemented as a type class, which was originally used to implement extensible records (Kiselyov et al. 2004). Because no recompilation of the library modules is needed to allow case analysis over a new type, this library scores good on this criterion.

MultiRec supports ad-hoc definitions by means of case analysis on the type representation. Since MultiRec is not tied to a single type representation datatype but allows as many as needed, modifying the Company type representation does not require recompilation of the MultiRec library and unrelated functions. However, ad-hoc cases are inconvenient when MultiRec is used on polymorphic types. Each polymorphic instance requires a different, redundant type representation datatype. Therefore MultiRec only scores sufficient.

Ad-hoc definitions for constructors. Can we give an ad-hoc definition for a particular constructor, and let the remaining constructors be handled generically? We take the function *rmWeights* as our test. If a library allows the implementation of this function such that an explicit case for the *WithWeight* constructor is given and the remaining constructors are handled generically, then that library scores good on this criterion.

The LIGD, Spine, and PolyLib approaches do not support ad-hoc definitions for datatypes. Implementing ad-hoc definitions for constructors requires ad-hoc cases. Therefore the first two approaches would require recompilation and the last one would only be usable with a regular datatype. Should we then declare that this criterion is unsupported? We do not think so. The user might be interested in providing an ad-hoc constructor definition, and still be willing to pay the price for the lack of support for ad-hoc definitions. We make this explicit in Figure 4.7: it is allowed to cheat in the *rmWeights* test for the ad-hoc definitions for datatypes criterion.

The LIGD, PolyLib, and Spine libraries support the definition of *rmWeights* as required. However, LIGD scores sufficient because of additional complications. As explained in Section 4.3.2, LIGD needs a modified datatype representation that allows ad-hoc definitions. This gives a total of two representations for WTree, one for generic extension and the other for non-generic extension.

The six approaches that support ad-hoc definitions for datatypes, also support ad-hoc definitions for constructors, hence SYB, SYB3, Uniplate, EMGM, Smash, RepLib and MultiRec score good on this criterion.

Extensibility. Can a programmer extend the universe of a generic function in a different module than that of the definition without the need for recompilation? Libraries that allow the extension of the generic *gshow* function with a case for printing lists score good. As mentioned before, extensibility is not possible for approaches that do not support ad-hoc cases. For this reason the LIGD, PolyLib, and Spine approaches score bad.

Before proceeding to the evaluation, let us remark that a library that supports ad-hoc cases can be made extensible. The trick is to extend the generic function with an argument that receives the ad-hoc case (or cases) with which the generic function is

extended. Such a trick would be possible with SYB, for example. However, this is unacceptable for two reasons. First, this would impose a burden on the user: the generic function has to be "closed" by the programmer before use. Second, functionality that is implemented on top of such an extensible generic function would have to expose the extension argument in its interface. An example of such functionality is discussed by Lämmel and Peyton Jones (2005) in their QuickCheck case study. QuickCheck implements shrinking of test data by using a *shrink* generic function, which should be extensible. If function extensibility would be implemented as proposed in this paragraph, the high-level *quickCheck* function would have to include extension arguments for *shrink*. For this reason, we do not accept such implementations of extensibility in our evaluation.

The SYB, Uniplate and MultiRec libraries support ad-hoc definitions, but do not support extensible generic functions. Therefore they score bad on this criterion.

In SYB3, EMGM, and RepLib, ad-hoc cases are given in instance declarations, which can reside in separate modules. Therefore these libraries support extensible generic functions.

In Smash, the traversal for a particular strategy can be overridden for a type. This overriding takes place in a separate module, so Smash supports extensible generic functions.

Multiple arguments. Can a generic programming library support a generic function definition that consumes more than one generic argument, such as the generic equality function? If generic equality is definable then the approach scores good. If the definition is more involved than those of other consumer functions such as *gshow* and *selectSalary*, then the approach scores sufficient.

The LIGD, PolyLib, EMGM, RepLib, and MultiRec approaches support the definition of generic equality. Furthermore, equality is not more difficult to define than other consumer functions. For example, in LIGD, every case of equality matches a type representation and two structurally represented values that are to be compared. Because these two values have the same type, usual pattern matching is enough to give the definition. Defining generic equality is not any more difficult than defining *gshow*. Therefore, these approaches score good on this criterion.

The definition of equality in Spine is not more complex than other consumer functions. Therefore Spine scores good. It can be argued, however, that the Spine version is not entirely satisfactory because it ultimately relies on equality of constructor names. Therefore the user could make a mistake when providing a constructor name in the representation.

The SYB library only scores sufficient on this criterion. The reason is that the definition is not as direct as for other functions such as *gshow* and *selectSalary*. While the Spine definition *equalSpine* can inspect the two arguments to be compared, in SYB the *gfoldl* combinator forces to process one argument at a time. For this reason, the definition of generic equality has to perform the traversal of the arguments in two stages. The definition can be found in Lämmel and Peyton Jones (2004)

	LIGD	PolyP	SYB	SYB3	Spine	EMGM	RepLib	Smash	Uniplate	MultiRec
geq	28.19	5.97	52.31	86.00	15.03	1.44	7.39	4.00	-	6.50
selectInt	25.00	-	36.00	15.06	13.69	23.25	12.94	14.63	5.81	47.44
rmWeights	3.32	1.00	68.68	5.89	1.85	3.52	3.57	1.83	3.94	1.69

Figure 4.9: Preliminary performance experiments.

Smash supports multiple arguments to a generic function essentially through currying – a special traversal strategy that traverses several terms in lock-step. However, the fact that a special purpose traversal must be used for functions on multiple arguments imposes a burden on the user. The user has to write one more structure representation per datatype. Therefore Smash only scores sufficient.

The traversal combinators of the Uniplate library are designed for single argument consumers. We have not been able to write a generic equality function for this approach, so Uniplate scores bad.

Constructor names. Is the approach able to provide the names of the constructors to which the generic function is applied? Library approaches that support the definition of *gshow* score good.

With the exception of Uniplate, all generic programming libraries discussed in this chapter provide support for constructor names in their structure representations. The Uniplate library itself does not provide any means to access constructor names.

Consumers, transformers, and producers. Generic libraries that can define functions in the three categories: consumers, transformers and producers, score good. This is the case for LIGD, PolyLib, EMGM, RepLib, and MultiRec.

If a library uses a different structure representation or type representation for say consumer and producer functions, that library scores sufficient. This is the case of SYB, SYB3, Smash, and Spine. Furthermore, Smash uses a different representation (traversal strategy) for transformers than for consumers, therefore it scores sufficient as well on that criterion. Why is it a disadvantage to have several structure representations? Because this implies more work for the programmer when doing generic universe extension: more representations have to be written per datatype.

The Uniplate library does not contain combinators to write producer functions, so it scores bad.

Performance. We have used some of the test functions for a performance benchmark but the results are very sensitive to small code differences and compiler optimizations so firm conclusions are difficult to draw. These tests were compiled using GHC 6.8.3 using the optimization flag `-O2`. As a sample, Figure 4.9 shows running times (in multiples of the running time of a hand-coded monomorphic version) for the *geq*, *selectInt* and *rmWeights* tests.

For the *geq* test, the compiler manages to produce very efficient code for EMGM, while the SYB family has problems with the two-argument traversal. Uniplate produces the best result for the *selectInt* test. However, for such results, the programmer is expected to manually define a large number of structure representations (as many as the number of datatypes squared). For *rmWeights*, there are several approaches that are within a factor of two of the hand written approach. PolyP shows the most impressive result: the compiler produces code as efficient as that of the hand-written version. The best overall performance score is shared between PolyP, EMGM, RepLib, Smash, and Uniplate.

Portability. Figure 4.10 shows that the majority of approaches rely on compiler extensions provided by GHC to some extent. Approaches that are most portable rely on few extensions or on extensions that are likely to be included in Haskell Prime (Peyton Jones et al. 2007).

Of all generic programming libraries, LIGD, EMGM, and Uniplate are the most portable. The only compiler extension that LIGD relies on is existential types, and this extension is very likely to be included in the Haskell Prime. However, LIGD needs rank-2 types for the representations of GRose and NGRose, but not for other representations or functions. Hence rank-2 types are not an essential part of the LIGD approach.

The EMGM library relies on multi-parameter type classes (also likely to be included in the next standard) to implement implicit type representations. This type class makes EMGM more convenient to use, but, even without it, it would still be possible to do generic programming in EMGM, be it in a less convenient way.

In SYB3, overlapping and undecidable instances are needed for the implementation of extensibility and ad-hoc cases. Overlapping instances arise because of the overlap between the generic case and the type-specific cases. Undecidable instances are required for the Sat type class, which is used to implement abstraction over type classes. This library and its predecessor (SYB) use rank-2 polymorphism to define the *gfoldl* and *gunfoldl* combinators. These two libraries, as well as RepLib, use *unsafeCoerce* to implement type safe casts.

GADTs are an essential building block of Spine, RepLib and MultiRec. These are used to represent types and their structure. Moreover, the first two libraries also make use of existential types.

The SYB3, RepLib and MultiRec libraries provide automatic generation of representations, which is implemented using Template Haskell. The SYB library, on the other hand, relies on compiler support for deriving Data and Typeable instances.

Uniplate can be defined in Haskell 98. However, in order to use multi-type traversals, multi-parameter type classes are needed. Uniplate can derive representations by either using a tool that uses Template Haskell, or by relying on compiler support to derive Data and Typeable. However, the use of these extension is optional, because structure representations can be written by programmers directly.

Smash relies on various extensions such as multi-parameter type classes, undecid-

able instances, overlapping instances, and functional dependencies. This is needed to implement the techniques for handling ad-hoc cases and traversal strategies.

The implementation of MultiRec also uses type families to map datatypes to their respective representations.

Overhead of library use. How much overhead is imposed on the programmer for the use of a generic programming library? We are interested in the following aspects:

- *Automatic generation of representations.* The libraries that offer support for automatic generation of representations are SYB, SYB3, EMGM, Uniplate, RepLib and MultiRec.

  The SYB library relies on GHC to generate Typeable and Data instances for new datatypes. The SYB library scores good on this criterion.

  The RepLib library includes Template Haskell code to generate structure representations for new datatypes in its distribution. However, RepLib does not support the generation of representations for arity two generic functions and does not include the machinery for such representations. Furthermore automatic generation fails when a type synonym is used in a datatype declaration. RepLib scores sufficient because of the problems mentioned.

  MultiRec also includes automatic generation of structure representations based on Template Haskell. Unfortunately, generation does not work for polymorphic datatypes such as lists, and in those cases the user must manually define structure representations. MultiRec only scores sufficient in this criterion.

  The SYB3 library also uses Template Haskell to generate representations. However, the generated representations for BinTree cause non-termination when type-safe casts are used on a BinTree value. This is a serious problem: regular datatypes and type-safe casting are very commonly used. Hence this approach does not score good but sufficient.

  Uniplate can use the Typeable and Data instances of GHC for automatic generation of representations. Furthermore, a separate tool, based on Template Haskell, is provided to derive instances. The Uniplate library scores good on this criterion.

  PolyLib used to include support for generation of representations, but this functionality broke with version 2 of Template Haskell.

  Note that automatic generation of representations in all approaches is limited to datatypes with no arguments of higher-kinds, hence GRose and NGRose are not supported.

- *Number of structure representations.* PolyLib only supports regular datatypes of kind $\star \to \star$, thus it only has one representation. It would be straightforward to add a new representation for each kind, but it would still only support regular datatypes. MultiRec is equipped with one representation only.

The LIGD, EMGM, and RepLib libraries have two sorts of representations: (1) a representation for $\star$-types (for example Rep in Section 4.2), and (2) representations for type constructors, which are arity-based (Section 4.3.3). The latter consists of a number of arity-specific representations. For example, to write the *gmap* function we would have to use a representation of arity two. Which arities should be supported? Probably the best answer is to support the arities for which useful generic functions are known: *crush* (arity one), *gmap* (arity two), and generic zip (arity three). This makes a total of four representations needed for these libraries, one to represent $\star$-types, and three for all useful arities.

Note, however, that functions of arity one can be defined using a representation of arity three. This means that we could remove representations of arities one and two. So, we could imagine a library needing only two representations. The next step is to subsume the representation for $\star$-types with the arity three representation. Although this makes some approaches more inconvenient – for instance, the EMGM approach would lose the generic dispatcher. Although reducing the number of representations is possible, we do not do so in this comparison, because it is rather inelegant. Defining generic equality using a representation of arity three would leave a number of unused type variables that might make the definition confusing.

When a new datatype is used with SYB/SYB3, the structure representation is given in a Data instance. This instance has two methods *gfoldl* and *gunfold* which are used for consumer and transformer, and producer generic functions respectively. Therefore every new datatype needs two representations to be used with SYB/SYB3 functions.

The Spine library needs three structure representations per datatype. The first, the spine representation, is used with consumer and transformer functions. And the second, the type spine view, is used with producer functions. The third representation is used to write generic functions that abstract over type constructors of kind $\star \to \star$.

In Uniplate, the number of structure representations that are needed can range from one Uniplate instance per datatype, to $O(n^2)$ instances for a system of n datatypes, when maximum performance is wanted. For our comparison, we assume that one representation is needed.

Smash is specifically designed to allow arbitrary number of custom traversal strategies. Although three strategies are fundamental – reconstruction, reduction, and parallel traversal – the simplified variations of these turn out more convenient and frequently used. However, this also means that the programmer defines more structure representations than in other libraries. The representations that are used in this evaluation are eight: a rewriting strategy, a reduction strategy, a reduction strategy with constructor names, a twin traversal strategy, a shallow reduction traversal, two traversals for abstracting over type constructors of kinds $\star \to \star$ and $\star \to \star \to \star$, and a traversal strategy for producer functions.

- *Work to instantiate a generic function.* Ideally, having the definition of a generic function and the structure representation for a datatype should be sufficient for applying the generic function. In this case, the total amount of work amounts to defining the structure representation and the generic function.

  The LIGD, Spine and MultiRec approaches score good because they allow direct use of a generic function on a type. All what is needed is to apply the generic function to the appropriate type representation. Type class based approaches, such as SYB, Smash, Uniplate, and SYB3 require even less effort because the type representation passed to the generic function is implicit.

  In contrast, the EMGM and RepLib libraries require an instance declaration that enables the use a generic function on a datatype. In addition to defining generic functions and structure representation, the user must also spend additional effort defining instances for every use of a generic function on a datatype. On the other hand, this allows precise control over the domain of a generic function, which fulfills the subuniverses criterion.

- *Work to define a generic function.* Is it possible to quickly write a simple generic function? A library scores good if it requires roughly one definition per generic function case, with no need for additional artefacts.

  The LIGD, SYB3, and RepLib libraries score bad on this criterion. In LIGD, the definitions of generic functions become more verbose because of the emulation of GADTs. However, this overhead does not arise in implementations of LIGD that use GADTs directly. In RepLib we need to define a dictionary datatype, and an instance of the `Sat` type class, in addition to the type class definition that implements the generic function. In SYB3 the definitions needed (besides a type class for the function) are the dictionary type, a `Sat` instance for the dictionary, and a dictionary proxy value. Therefore these libraries only score sufficient. The other libraries score good because they demand less "encoding work" when defining a generic function.

Practical aspects. For this criterion we consider aspects such as

- there is a library distribution available online,
- the library interface is well-documented,
- and other aspects such as an interface organized into modules, and common generic functions.

The LIGD and Spine libraries do not have distributions online. These libraries score bad. PolyLib has an online distribution (as part of PolyP version 2) but the library is not maintained anymore.

The SYB library is distributed with the GHC compiler. This distribution includes a number of traversal combinators for common generic programming tasks and Haddock

documentation. The GHC compiler supports the automatic generation of Typeable and Data instances. This library scores good.

The official SYB3 distribution failed to compile with the versions of GHC that we used in this comparison (6.6, 6.8.1, 6.8.2). This distribution can be downloaded from: `http://homepages.cwi.nl/~ralf/syb3/code.html`. There is an alternative distribution of this library which is available as a Darcs repository from: `http://happs.org/HAppS/syb-with-class`. This distribution is a cabal package that includes Haddock documentation for the functions that it provides. However, it does not contain many important combinators such as *gmapAccumQ* and *gzipWithQ* amongst others.

The EMGM, RepLib, Uniplate and MultiRec libraries have online distributions at the HackageDB web site. All libraries all well-structured, have a number of useful predefined functions, and come with Haddock documentation. All these libraries score good.

The Smash library is distributed as a stand-alone module that can be downloaded from `http://okmij.org/ftp/Haskell/generics.html#Smash`. There are a few example functions in that module that illustrate the use of the approach. However, the library is not as well structured or documented as other generic programming libraries.

Implementation mechanisms. What are the mechanisms through which a library encodes a type or its representation? We have the following options:

- *Types and structure represented by GADTs.* Types are encoded by a representation GADT, where each type is represented by a constructor. The GADT may also have a constructor which encodes datatypes structurally (like *RType* in this chapter).

- *Types and structure represented by datatypes.* When GADTs are not available, it is still possible to emulate them by embedding conversion functions in the datatype constructors. In this way a normal datatype can represent types and their structure.

- *Rank-2 structure representation combinators.* Yet another alternative is to represent the structure of a datatype using a rank-2 combinator such as *gfoldl* in SYB.

- *Type safe cast.* Type safe casts are used to implement type-specific functionality, or ad-hoc cases. Type safe casting attempts to convert `a`-values into `b`-values at runtime. Statically it may not be known that `a` and `b` are the same type, but if at runtime it is the case, the conversion succeeds.

- *Extensible records and type-level evaluation.* The work of Kiselyov et al. (2004) introduces various techniques to implement extensible records in Haskell. The techniques make advanced use of type classes to perform record lookup statically. This operation is an instance of a general design pattern: encoding type-level computations using multi-parameter type classes and functional dependencies.

	LIGD	PolyLib	SYB	SYB3	Spine	EMGM	RepLib	Smash	Uniplate	MultiRec
Implementation mechanisms										
Type representation is GADT					●		●			●
Type representation is D.T.	●									
Rank-2 struc. repr. combinator			●	●						
Type safe cast			●	●			●			
Extensible rec. and type eval.								●		
Type classes		●	●	●		●	●	●	●	●
Type families										●
Abstraction over type classes				●			●			
Compiler extensions										
Undecidable instances		●		●				●		
Overlapping instances				●				●		
Multi-parameter type classes		●				●		●	●	●
Functional dependencies								●		
Type families										●
Rank-2 polymorphism			●	●						●
Existential types	●				●		●			
GADTs					●		●			●
unsafeCoerce			●	●			●			
Template Haskell				●			●		●	●
Derive Data and Typeable			●						●	
Views										
Fixed point view		●								●
Sum of products	●					●				
Spine			●	●	●					
Lifted spine					●					
RepLib							●			
Uniplate									●	
Smash								●		

Figure 4.10: Implementation mechanisms, required compiler extensions, and views.

- *Type classes.* Type classes may be used in a variety of ways: to represent types and their structure and perform case analysis on them, to overload structure representation combinators, and to provide extensibility to generic functions.

- *Type families.* Type families are used to map a datatype or a system of datatypes to their corresponding structure representation.

- *Abstraction over type classes.* Generic programming libraries that support extensible generic functions, do so by using type classes. Some of this approaches, however, require a form of abstraction over type classes, which can be encoded by a technique that uses explicit dictionaries, popularized by Lämmel and Peyton Jones (2005).

The LIGD and Spine libraries represent types and structure as datatypes and GADTs respectively, while EMGM uses type classes to do so.

In SYB and SYB3, datatypes are structurally represented by rank-2 combinators *gfoldl* and *gunfold*.

Ad-hoc cases in SYB are given using pre-defined combinators such as *extQ* and *mkQ*, which are implemented using type safe casts.

In SYB3, case analysis over types is performed directly by the type class system, because generic functions are type classes. However, type safe casts are still important because they are used to implement functions such as equality. Furthermore, this approach implements abstraction over type classes to support extensibility.

The RepLib library is an interesting combination. It has a datatype that represents types and their structure, and so generic functions are defined by pattern matching on those representation values. On the other hand, RepLib also uses type classes to allow the extension of a generic function with a new type-specific case. To allow extension type classes must encode type class abstraction using the same technique as in SYB3. Optionally, the RepLib library allows the programmer to use a programming style reminiscent of SYB, where ad-hoc cases are aggregated using functions such as *extQ* and *mkQ*. These combinators are implemented using the GADT-based representations and type safe casts.

The Smash library uses an extensible record-like list of functions to encode ad-hoc cases. Case analysis on types is performed by a lookup operation, which in turn is implemented by type-level type-case. This library also uses type-level evaluation to determine the argument and return types of method *gapp*, based on the traversal strategy and the list of ad-hoc cases.

MultiRec is the only approach that uses type families. Type families map a type that represents a system of datatypes into a structure representation. This representation is built out of pre-defined GADT type constructors and associated to individual datatypes in the system by means of multi-parameter type classes.

Views. What are the views that the generic library supports? That is, how are datatypes encoded in structure representations and what are the view types used in them?

The LIGD and EMGM libraries encode datatypes as sums of products, where the sums encode the choice of a constructor and the products encode the fields used in them. This view is usually referred to as sum of products. The representations for sums of products that are based on arities (Section 4.3.3) can be used to abstract over type constructors.

PolyP represents the structure of regular datatypes using a fixed point view. This view makes use of sums and products to encode constructors and their arguments, but additionally the use of a type-level fixed point constructor the recursive occurrences of the datatype are made explicit. MultiRec extends the fixed point view to represent systems of mutually recursive datatypes.

The Spine approach uses the Spine datatype to make the application of a constructor to its arguments observable to a generic function. As observed by the authors of this view, the *gfoldl* combinator used in SYB and SYB3 is a catamorphism of a Spine value, and hence these approaches also use the spine view. The SYB, SYB3, and Spine approaches also provide a type-spine view that is used to write producer generic functions. Unlike SYB and SYB3, Spine supports abstraction over $\star \longrightarrow \star$-types due to an additional view called the lifted Spine view.

The RepLib library also has a different view, datatypes are represented as a list of constructor representations, which are a heterogeneous list of constructor arguments. Like in LIGD and EMGM, the structure representation can be adapted to be arity based, in order to support abstraction over type constructors.

The Uniplate library also has a view of its own. All traversal combinators in this library are based on the *uniplate* operation. This operation takes an argument of a particular datatype, and returns its children that have the same type as the argument, and a function to reconstruct the argument with new children.

It is difficult to point to specific views in the Smash library. Although there are three basic strategies (rewriting, reduction, and lock-step reduction), the rest of the strategies are not defined using any of the more fundamental ones. Hence every traversal strategy can be considered as a way to view the structure of a datatype.

Ease of learning and use. It is hard to determine how easy it is to learn using a generic programming library. We approximate this criterion by looking at the complexity underlying the mechanisms used in the implementation of libraries. Below we describe the difficulties that arise with some of the implementation mechanisms:

- *Rank-2 structure representation combinators.* There are two problems with rank-2 structure representation combinators. First, rank-2 polymorphic types are difficult to understand for beginning users. This implies that defining a generic function from scratch – that is, using the type structure directly, bypassing any pre-defined combinators – presents more difficulties than in other approaches. Second, structure representation combinators such as those used in SYB can be used directly to define functions that consume one argument. But if two arguments are to be consumed instead (which is the case in generic equality), then the definition of the function becomes complicated.

- *Extensible records and type-level evaluation.* The techniques to encode extensible records make advanced use of type classes and functional dependencies. This encoding can be troublesome to the beginning generic programmer in at least one way: type errors arising from such programs can be very difficult to debug.

- *Abstraction over type classes.* Abstraction over type classes is emulated by means of explicit dictionaries that represent the class that is being abstracted. This is an advanced type class programming technique and it might confuse beginning generic programmers.

- *Arity based representations.* Arity based representations are used to represent type constructors. However, it is more difficult to relate the type signature of an arity-based generic function to that of an instance. For example, generic map has type $\mathsf{Rep2\ a\ b} \to \mathsf{a} \to \mathsf{b}$, which does not bear a strong resemblance to the type of the BinTree instance of map: $(\mathsf{a} \to \mathsf{b}) \to \mathsf{BinTree\ a} \to \mathsf{BinTree\ b}$. For this reason, programming with arity-based representations might be challenging to a beginner.

The approaches that suffer from the first difficulty are SYB and SYB3. The second difficulty affects Smash. The third difficulty affects SYB3 and RepLib. The fourth difficulty affects LIGD, EMGM, and RepLib. However, the first two libraries, need such arities only occasionally, namely to define functions such as *gmap* and *crushRight*.

Another problem that can impede learning and use of an approach is the use of a relatively large number of implementation mechanisms. This is the case for SYB3, RepLib and MultiRec. While it is possible to work out how one of the many mechanisms, for example GADTs in RepLib, is used into writing a generic function, it is much more difficult to understand the interactions of all the mechanisms in the same library. This, we believe, will make it more difficult for new users to learn and use SYB3, RepLib and MultiRec.

4.8 Conclusions

We have introduced a set of criteria to compare libraries for generic programming in Haskell. These criteria can be viewed as a characterisation of generic programming in Haskell. Furthermore, we have designed a generic programming benchmark: a set of characteristic examples that check whether or not criteria are supported by generic programming libraries. Using the criteria and the benchmark, we have compared ten approaches to generic programming in Haskell.

Is it possible to combine the libraries into a single one that has a perfect score? Our comparison seems to suggest otherwise. A good score on one criterion generally causes problems in another. For example, approaches with extensible generic functions sometimes have problems that are absent in non-extensible ones. The SYB3 library is extensible but defining a generic function requires more boilerplate than in SYB. Furthermore, SYB3 has a smaller universe than SYB. And while the EMGM library

provides extensible generic functions, defining higher-order generic functions is far from trivial.

What is the best generic programming library? Since no library has good scores on all criteria, the answer depends on the scenario at hand. Some libraries, such as LIGD, PolyLib and Spine, score bad on important criteria such as ad-hoc cases, separate compilation and universe size (support for mutual recursion), and are unlikely candidates for practical use. We now discuss which libraries are most suitable for implementing one of three typical generic programming scenarios.

Consider the criteria required for transformation traversals. Implementing traversals over abstract syntax trees requires support for mutually recursive datatypes. Furthermore, traversals are higher-order generic functions since they are parametrized by the actual transformations. Traversals also require little work to define a generic function, because users often define their own traversals. The libraries that best satisfy these criteria are SYB and Uniplate. Uniplate does not support higher-orderness, but Uniplate functions are monomorphic, so that criterion is not needed. If extensible traversals are needed and the additional work to define a generic function is not a problem, we can also use SYB3 or RepLib.

The criterion needed for operating over the elements of a container is abstraction over type constructors. Ad-hoc cases are also commonly needed to process a container in a particular way. The libraries that best fit this scenario are EMGM and RepLib. Smash can also be used but the high number of representations may demand more effort from the user.

For serialization, one can use approaches that have good scores on constructor names, producers, ad-hoc cases and universe size (mutual recursion), namely EMGM and RepLib. SYB, SYB3 and Smash can also be used, if one is willing to learn a different API for writing a producer function.

Acknowledgements. This research has been partially funded by the Netherlands Organisation for Scientific Research (NWO), via the Real-life Datatype-Generic programming project, project nr. 612.063.613. We thank J. Gibbons, S. Leather and J.P. Magalhães for their thoughtful comments and suggestions. We also thank the participants of the generics mailing list for the discussions and the code examples that sparked the work for this chapter. In particular, S. Weirich and J. Cheney provided some of the code on which our test suite is based. Finally, Andres Löh provided useful comments and formatting tips.

5 Enumerating Well-Typed Terms Generically

In this chapter we address the problem of generating well-typed terms using generic programming. Well-typed terms are often encoded by generalized algebraic datatypes (GADTs) and often also with existential types. The Spine approach (Hinze et al. 2006; Hinze and Löh 2006) to generic programming supports GADTs, but unfortunately it does not support the definition of generic producers for existentials. We describe how to extend the Spine approach to support existentials and we use the improved Spine to define a generic enumeration function. We show that the enumeration function can be used to generate the terms of simply typed lambda calculus.

5.1 Introduction

Since their introduction to Haskell, Generalized Algebraic Datatypes (GADTs) (Xi et al. 2003; Cheney and Hinze 2003; Peyton Jones et al. 2006) have often been applied to improve the reliability of programs. GADTs allow the encoding of datatype invariants by type constraints in constructor signatures. With this information, the compiler rejects during the type-checking process values for which such invariants do not hold. In particular, GADTs can be used to model sets of well-typed terms such that values representing ill-typed terms cannot be constructed. Applications of GADTs include well-typed program transformations, implementation of dynamic typing, staged computation, ad-hoc polymorphism and tag-less interpreters.

Given the growing relevance of GADTs, it is important to provide generic programming support for generalized datatype definitions. The generation of datatype values using generic programming is of particular interest. Generic value generation has been used before to produce test data against which to check the validity of program properties (Koopman et al. 2003). In generic value generation, the datatype definition acts as a specification. However, this specification is often imprecise since it gives rise to either values that do not occur in practice, or, worse, ill-formed values (for example, a program fragment with unbound variables). For this reason, the QuickCheck system (Claessen and Hughes 2000) allows the definition of custom generators.

GADT definitions are more precise than normal datatypes. In the case of well-typed terms, the constraints in the datatype definition describe the formation rules of a well-typed value. It follows that a generic producer function that takes a GADT definition as a starting point, should produce values that are better suited for testing program properties. For example, it should be possible to use a generic value generation function

with a GADT encoding lambda calculus, in order to produce a well-typed lambda term and test with it a tag-less interpreter.

The *spine* view (Hinze et al. 2006; Hinze and Löh 2006), which is based on "Scrap Your Boilerplate" (Lämmel and Peyton Jones 2003), is the only approach to generic programming in Haskell that supports GADTs. The main idea behind the spine view is to make the application of a data constructor to its arguments explicit. The spine view represents a datatype value by means of two cases: the representation of a datatype constructor, and the representation of the application to constructor arguments. A generic function can then be defined by case analysis on the spine view. In Hinze and Löh (2006), the authors describe how to use the spine view to represent GADT values and define generic functions to consume and produce such values.

Besides GADT definitions, well-typed terms often make use of existentially quantified type variables. For example, the type of expression application is that of the function return type. The argument type is usually hidden from the application type and is therefore represented by existential quantification. Under certain conditions, the spine view supports the definition of generic functions that *consume* existentially typed values. Unfortunately, it cannot be used to define a generic function that *produces* them. It follows that the spine view cannot in general be used to define generic enumerators for well-typed terms.

This chapter extends the spine view to allow the use of producer functions on existential types. We make the following contributions:

- We show how to support existential types systematically within the spine view. We extend the spine view to encode existentially quantified type variables explicitly. This enables the definition of generic functions that perform case analysis on such types. As a consequence, the extended spine view supports the definition of generic producers that work on existential types. We demonstrate the increased generality by defining generic serialization and deserialization for existential types and GADTs.

- We define a generic enumeration function that can be used with GADTs and existential types. This function can be used to enumerate the well-typed terms represented by a GADT. Consider a GADT that represents terms in the simply typed lambda calculus. The enumeration of terms with type $\mathsf{Expr}\ ((\mathsf{b} \to \mathsf{c}) \to (\mathsf{a} \to \mathsf{b}) \to \mathsf{a} \to \mathsf{c})$ yields the term that corresponds to function composition. The enumeration function that we introduce requires explicit support of existential types in producers. Therefore, prior to the work in this chapter, it could not be defined in other approaches such as that of Hinze and Löh (2006).

This chapter is organized as follows. Section 5.2 introduces the spine view and gives several examples illustrating why this view is not suitable to define producers for existential types. Section 5.3 describes our extensions to the spine view, which enable producer support for existential types. Section 5.4 uses the extended spine view to define a generic enumeration function. The enumeration function is then used to produce

well-typed lambda calculus terms. Section 5.5 compares the work in this chapter with related work. Section 5.6 concludes.

5.2 The spine view

The spine view was introduced by Hinze et al. (2006). This view supports the definition of consumer and transformer generic functions. We introduce the spine view using the generic show function as an example. This function prints the textual representation of a value based on the type structure encoded by the view. To implement this function, we need case analysis on types to implement type-dependent behaviour.

5.2.1 Case analysis on types

The spine view uses GADTs to implement case analysis on types. We define a type representation datatype where each constructor represents a specific type:

```
data Type :: ∗ → ∗ where
  Int      :: Type Int
  Maybe    :: Type a → Type (Maybe a)
  Either   :: Type a → Type b → Type (Either a b)
  List     :: Type a → Type [a]
  (:→)     :: Type a → Type b → Type (a → b)
```

An overloaded function can be implemented by performing case analysis on types. To perform case analysis on types we pattern match on the type representation values. The GADT pattern matching semantics (Peyton Jones et al. 2006) ensures that the type variable a is refined to the target type of the matched constructor:

```
show :: Type a → a → String
show Int          n          = showInt n
show (Maybe a)    (Just x)   = paren ("Just"  • show a x)
show (Maybe a)    Nothing    = "Nothing"
show (Either a b) (Left x)   = paren ("Left"  • show a x)
show (Either a b) (Right y)  = paren ("Right" • show b y)
show (List a)     ((:) x xs) = paren ("(:)"   • show a x • show (List a) xs)
show (List a)     []         = "[]"
```

This function prints a textual representation of a datatype value. Note that we choose to print lists in prefix syntax rather than the usual Haskell notation. The operator (•) separates two strings with a white space, and *paren* prints parentheses around a string argument.

5.2.2 The spine representation of values

Type representations can be used to implement overloaded functions, but such functions are not generic. The user needs to define new *show* cases for every datatype added to the program. To define generic functions, we make use of the spine view.

The spine view represents all datatype values by means of two cases: a constructor and the application of a (partially applied) constructor to an argument. This is embodied in the Spine datatype:

```
data Spine :: * → * where
  Con  :: ConInfo a → Spine a
  (:◇:) :: Spine (a → b) → Typed a → Spine b
infixl 0 :◇:
```

The *Con* case of the spine view stores a value constructor of type a together with additional information including the constructor name, the fixity, and the constructor tag. This additional information is stored in the datatype ConInfo. The application case (:◇:) consists of a functional value Spine that consumes a-values, and the argument a paired with its type representation in the datatype Typed. We show Typed, and a simplified ConInfo containing only the constructor name below:

```
data ConInfo a = ConInfo{conName :: String, conVal :: a}
data Typed a   = (:▷:)    {val :: a, rep :: Type a}
```

To write a generic function, we first convert a value to its Spine representation. We show how to perform this conversion using the type-indexed function *toSpine*:

```
toSpine :: Type a → a → Spine a
toSpine Int          x          = Con (conint x    x)
toSpine (Maybe a)    (Just x)   = Con (conjust     Just)     :◇: x :▷: a
toSpine (Maybe a)    Nothing    = Con (connothing Nothing)
toSpine (Either a b) (Left x)   = Con (conleft     Left)     :◇: x :▷: a
toSpine (Either a b) (Right y)  = Con (conright    Right)    :◇: y :▷: b
toSpine (List a)     ((:) x xs) = Con (concons     (:))      :◇: x :▷: a
                                                             :◇: xs :▷: List a
toSpine (List a)     []         = Con (connil      [])
```

Because we reuse the constructor information in later sections of the chapter, we define ConInfo values separately. We give some examples below:

```
conint :: Int → a → ConInfo a
conint i    = ConInfo (showInt i)

connothing, conjust :: a → ConInfo a
connothing = ConInfo "Nothing"
conjust    = ConInfo "Just"
```

In summary, to enable generic programming using the spine view, we have defined a GADT for type representations, the Spine datatype, and conversions from datatype values to their spine representations. The conversions for datatypes have to be written only once, and then the same conversion can be reused for different generic functions. The conversion to the spine representation is regular enough that it can be automatically generated from the syntax trees of datatype declarations[1].

Equipped with the spine representation, we can write a number of generic functions. For example, this is the definition of generic *show*:

$$
\begin{aligned}
&\mathit{show} :: \mathsf{Type\ a} \to \mathsf{a} \to \mathsf{String} \\
&\mathit{show}\ \mathit{rep}\ x = \mathit{paren}\ (\mathit{gshow}\ (\mathit{toSpine}\ \mathit{rep}\ x)) \\
&\mathit{gshow} :: \mathsf{Spine\ a} \to \mathsf{String} \\
&\mathit{gshow}\ (\mathit{Con}\ \mathit{con}) \quad = \mathit{conName}\ \mathit{con} \\
&\mathit{gshow}\ (\mathit{con} :\diamond: \mathit{arg}) = \mathit{gshow}\ \mathit{con} \bullet \mathit{show}\ (\mathit{rep}\ \mathit{arg})\ (\mathit{val}\ \mathit{arg})
\end{aligned}
$$

This function is a simplified variant of the *show* function defined in the Haskell prelude. All datatype values are printed uniformly: constructors are separated from the arguments by means of the $\bullet$ operator, and parentheses are printed around fully applied constructors.

5.2.3 Transformer functions

The spine view also supports the definition of generic transformer functions. Examples of such functions include incrementing all Int values in a tree, and applying a function to all nodes of a specific type in a tree.

To write such a function, we need to convert the spine representation back to the represented value *after* it has been traversed and transformed. This is achieved by the *fromSpine* function:

$$
\begin{aligned}
&\mathit{fromSpine} :: \mathsf{Spine\ a} \to \mathsf{a} \\
&\mathit{fromSpine}\ (\mathit{Con}\ \mathit{con}) \quad = \mathit{conVal}\ \mathit{con} \\
&\mathit{fromSpine}\ (\mathit{con} :\diamond: \mathit{arg}) = \mathit{fromSpine}\ \mathit{con}\ (\mathit{val}\ \mathit{arg})
\end{aligned}
$$

See Hinze et al. (2006) for examples of transformer functions using the spine view.

5.2.4 A view for producers

It is impossible to write *read*, the inverse to *show* using the current Spine datatype. We could for example use the following type for *read*:

$$\mathit{read} :: \mathsf{Type\ a} \to \mathsf{String} \to [(\mathsf{a}, \mathsf{String})]$$

This function produces all possible parses of type a (paired with unused input) from a representation for the type a and an input string. To write such a generic function,

[1] At the time of writing, Template Haskell cannot handle GADT declarations. Our prototype generates spine representations for GADTs independently.

we would need a spine representation to guide the parsing process. Unfortunately, a representation Spine a cannot be used for this purpose. A value of Spine a represents a particular value of type a (for example, a singleton list) rather than the full datatype structure (a description of the cons and nil constructors and their arguments). To enable a generic definition of generic read and other producer generic functions, Hinze and Löh (2006) introduce the type spine view. A view that describes all values of a rather than one in particular.

```
type TypeSpine a = [Signature a]
data Signature :: ∗ → ∗ where
  Sig    :: ConInfo a → Signature a
  (:⊕:) :: Signature (a → b) → Type a → Signature b
infixl 0 :⊕:
```

Here we have again two cases, one for encoding a constructor and another for the application of a (partially) applied constructor to an argument. The application case contains only a type representation and no argument value anymore. A value of TypeSpine a is a list of constructor signatures representing all constructors of the represented datatype. The type-indexed function *typeSpine* produces the type spine representations of all datatypes on which generic programming is to be used.

```
typeSpine :: Type a → TypeSpine a
typeSpine Int          = [Sig (conint i     i) | i ← [minBound..maxBound]]
typeSpine (Maybe a)    = [Sig (connothing Nothing)
                         ,Sig (conjust    Just)   :⊕: a]
typeSpine (Either a b) = [Sig (conleft    Left)   :⊕: a
                         ,Sig (conright   Right)  :⊕: b]
typeSpine (List a)     = [Sig (connil     [])
                         ,Sig (concons    (:))    :⊕: a :⊕: List a]
```

The generic parsing function, *read*, builds a parser that deserializes a value of type a:

```
read :: Type a → Parser a
```

For the purposes of this chapter, we assume that Parser is an abstract parser type with a monadic interface:

```
return :: a → Parser a
(>>=) :: Parser a → (a → Parser b) → Parser b
```

We also use some derived monadic operations such as application and sequencing:

```
ap    :: Parser (a → b) → Parser a → Parser b
(>>) :: Parser a → Parser b → Parser b
```

We also assume that we can build failing parsers and parse multiple alternatives:

```
noparse      :: Parser a
alternatives :: [Parser a] → Parser a
```

We also assume a couple of primitive parsers to recognize integers and Haskell tokens:

```
readInt :: Parser Int
lex     :: Parser String
```

The definition of generic read uses *readInt* to read an integer value. For other datatypes, we make parsers for each of the constructor representations and merge all the alternatives in a single parser.

```
read :: Type a → Parser a
read Int  = readInt
read rep = alternatives [readParen (gread conrep) | conrep ← typeSpine rep]
```

The generic parser of a constructor is built by induction on its signature representation. The base case (*Sig*) tries to recognize the constructor name and returns the constructor value. The application case parses the function and argument parts recursively and the results are combined using monadic application:

```
gread :: Signature a → Parser a
gread (Sig c)        = token (conName c) ≫ return (conVal c)
gread (con :⊕: arg) = gread con `ap` read arg
```

In the definitions of *read* and *gread*, we have used a couple of utility functions that consume a specific Haskell token and add parenthesis recognition around a parser argument.

```
token :: String → Parser ()
token s = do inp ← lex
             if inp ≡ s then return () else noparse
readParen :: Parser a → Parser a
readParen p = do token "("
                 res ← p
                 token ")"
                 return res
```

In the definition of generic read, we could also have used parser combinators based on an applicative interface (McBride and Paterson 2007) instead of a monadic one. For an example of parser combinators with an applicative interface see Swierstra and Duponcheel (1996). In Section 5.3.1 we show that existentially typed values cannot be parsed using purely applicative parser combinators, because generic read on existentials makes essential use of bind (≫=).

5.2.5 Generalized algebraic datatypes

Recall that generalized algebraic datatypes are datatypes to which type-level constraints are added. Such constraints can be used to encode invariants that datatype values must satisfy. For example, we can define a well-typed abstract syntax tree by having the syntactic categories of constructs in the target type of constructors:

```
data Expr :: ∗ → ∗ where
  EZero  :: Expr Int
  EFalse :: Expr Bool
  ESuc   :: Expr Int  → Expr Int
  ENot   :: Expr Bool → Expr Bool
  EIsZero :: Expr Int → Expr Bool
```

We have constants for integer and boolean values, and operators that act on them.

GADTs can easily be represented in the spine view. For instance, the definition of *toSpine* for this datatype is as follows:

```
toSpine :: Type a → a → Spine a
...
toSpine (Expr Int)  EZero       = Con (conezero  EZero)
toSpine (Expr Bool) EFalse      = Con (conefalse EFalse)
toSpine (Expr Int)  (ESuc e)    = Con (conesuc   ESuc)    :⋄: e :▷: Expr Int
toSpine (Expr Bool) (ENot e)    = Con (conenot   ENot)    :⋄: e :▷: Expr Bool
toSpine (Expr Bool) (EIsZero e) = Con (coneiszero EIsZero) :⋄: e :▷: Expr Int
```

This definition requires the extension of Type with the representation constructors *Expr* and *Bool*. Generic functions defined on the spine view, such as generic show, can now be used on Expr.

What about generic producer functions? These can be used on Expr too, because it is also possible to construct datatype representations for GADTs in the type spine view:

```
typeSpine :: Type a → TypeSpine a
...
typeSpine (Expr Int)  = [Sig (conezero  EZero)
                        ,Sig (conesuc   ESuc)    :⊕: Expr Int]
typeSpine (Expr Bool) = [Sig (conefalse EFalse)
                        ,Sig (conenot   ENot)    :⊕: Expr Bool
                        ,Sig (coneiszero EIsZero) :⊕: Expr Int]
```

To parse boolean expressions, we invoke the generic read function as follows:

```
readBoolExpr :: Parser (Expr Bool)
readBoolExpr = read (Expr Bool)
```

The parser for integer expressions would use a different argument for *Expr*. In this example, we are assuming that the expression to be parsed is always of a fixed type.

A more interesting scenario would be to leave the type of the GADT unspecified and let it be dynamically determined from the parsed value. This would be useful if the programmer wants to parse some well-typed expression regardless of the type that the expression has.

A possible solution to parsing a GADT without specifying its type argument would be to existentially quantify over that argument in the result of the parsing function. Next, we discuss how the spine view deals with existential types.

5.2.6 Existential types and consumer functions

In Haskell, existential types are introduced in constructor declarations. A type variable is existentially quantified if it is mentioned in the argument type declarations but omitted in the target type. For example, consider dynamically typed values:

```
data Dynamic :: ∗ where
  DynVal :: Type a → a → Dynamic
```

The type variable a in the declaration is existentially quantified. It is used to hide the type of the a-argument used when building a Dynamic value. The type a is kept abstract when pattern matching a Dynamic value, but by case analysing the type representation it is possible to dynamically recover the type a. Thus, statically, Dynamic values all have the same type, but, dynamically, the type distinction can be recovered and acted upon.

To represent dynamic values in Spine, we need to add type representations for Type itself and Dynamic. Hence, we add the following two constructors to Type:

```
data Type :: ∗ → ∗ where
  ...
  Type    :: Type a → Type (Type a)
  Dynamic :: Type Dynamic
```

Now, Dynamic values may be represented as follows by the spine view:

```
toSpine :: Type a → a → Spine a
...
toSpine Dynamic (DynVal rep val)
  = Con (condynval DynVal) :◇: rep :▷: Type rep :◇: val :▷: rep
```

While Dynamic values may be easily represented, this is not the case for all datatypes having existential types. Recall that in a spine representation, every constructor argument is paired with its type representation in the datatype Typed. In general, in constructors having existential types, it may not be possible to build such a pair because the representation of the existential type may be missing. The constructor *DynVal* is a special case, because it carries the representation type of the existential a. For an example where the representation of a constructor with existential types is not possible, consider adding an application constructor to the expression datatype:

```
data Expr :: ∗ → ∗ where
  ...
  EApp :: Expr (a → b) → Expr a → Expr b
```

and consider the corresponding *toSpine* alternative:

```
toSpine :: Type a → a → Spine a
...
toSpine (Expr b) (EApp fun arg)
  = Con (coneapp EApp) :◇: fun :▷: Expr (a :→ b) :◇: arg :▷: Expr a
```

This code is incorrect due to the unbound variable a which stands for the existential representation. The conclusion here is that the spine view can be used on an existential type, as long as the constructor in which it occurs carries a type representation for it.

5.2.7 Existential types and producer functions

The view for producer functions, the type spine view, cannot represent existential types as well as the spine view. For instance, consider how to generate such a representation for dynamic values:

```
typeSpine :: Type a → [Signature a]
...
typeSpine Dynamic = [Sig (condynval DynVal) :⊕: Type a :⊕: a]
```

What should the representation a be? There are two options, we either fix it to a single type representation or we range over all possible type representations. Choosing one type representation would be too restrictive, because *read* would only parse dynamic values of that type and fail on any other type. We try the second option:

```
typeSpine Dynamic = [Sig (condynval DynVal) :⊕: Type a :⊕: a | a ← types]
```

This code does not yet have the behaviour we desire. For *typeSpine* to be type-correct, *types* must return a list of representations all having the same type. Because Type is a singleton type (each type has only one value), *types* returns a single type representation. We would like *types* to generate a list of all possible type representations, but different type representations have different types. Therefore, *types* should return representations whose represented type is existentially quantified. To this end, we define the type of boxed type representations:

```
data BType = ∀a.Boxed (Type a)
applyBType :: (∀a.Type a → c) → BType → c
applyBType f (Boxed a) = f a
```

Now we can define the type spine of dynamic values, for which we assume a list of boxed representations (*types*):

```
types :: [BType]
typeSpine Dynamic
  = [Sig (condynval DynVal) :⊕: Type a :⊕: a | Boxed a ← types]
```

The boxed representations are used to construct a list of constructor signatures that represent a dynamic value of the corresponding type. There are infinitely many type instances of polymorphic types, therefore there are infinitely many Dynamic constructor representations. An infinite type spine is not a desirable representation to work with. The *read* function would try to parse the input using every Dynamic constructor representation. If there is a correct parse, parsing would eventually succeed with one of the representations. However, if there is no correct parse, parsing would not terminate. Moreover, this representation precludes implementing more efficient variants of parsing.

Infinite type spine representations for datatypes with existentials make the use of generic producers on such datatypes unpractical. Before describing a modified type spine view that solves this problem, we explore a couple of non-generic examples to motivate our design decisions.

We start with the parser definition for Dynamic values. In the code above, we are able to parse any possible dynamic value because there are *DynVal* constructor signatures for all possible types. For each signature, we build a parser that parses the corresponding type representation and a value having that type.

Now, rather than parsing the two arguments of the constructor *DynVal* independently, we introduce a dependency on representations. First, we parse the type representation for the existential. Then, we use it to build a parser of the corresponding type and parse the second argument. In this way, we no longer need to have an infinite representation of types because we obtain the representation of the existential during the parsing process:

```
read :: Type a → Parser a
...
read Dynamic = do
  Boxed a ← readType
  value   ← read a
  return (DynVal a value)
```

To this end, we use a function that parses type representations. Because the result may be of an arbitrary type, *readType* produces a representation that is boxed:

```
readType :: Parser BType
```

We defer the presentation of *readType* to Section 5.3.4.

The same technique can be used to parse any constructor having an existential type. For example, the definition for parsing expression applications is as follows:

```
read (Expr b) = do
  Boxed a ← readType
  fun     ← read (Expr (a :→ b))
  arg     ← read (Expr a)
  return (EApp fun arg)
```

In this example, the type representation that is parsed is used to build the type representations for the two remaining arguments.

These two examples show that constructors with existential types must be handled differently than other constructors. In such constructors, the constructor argument representations depend on the type representation of the existential type. In our examples, this dependency is witnessed by the dynamic construction of parsers based on the type representation that was previously parsed.

5.3 An improved Spine view: support for existential types

We start this section by showing how to extend the spine view for producers to represent existential types explicitly. Then, we show why this extension is also necessary for the consumer spine view.

5.3.1 The existential case for producer functions

We have learned two things from the *read* examples for constructors with existential types. First, we need a way to represent existential variables explicitly, so that generic functions can handle existential type variables specifically. And second, there is a dependency from constructor arguments on the existential variable. For example, we can only parse the function and argument parts of an expression application, if we have already parsed the existential type representation. We modify the type spine view to accommodate these two aspects. We extend the constructor signatures in this view with a constructor to represent existential quantification: *AllEx*. The dependency of type b on an existential type a is made explicit by means of a function from type representations of type a to representations of b.

```
data Signature :: ∗ → ∗ where
  Sig    :: a → Signature a
  (:⊕:) :: Signature (a → b) → Type a → Signature b
  AllEx :: (∀a. Type a → Signature b) → Signature b
```

Interestingly, the type variable a is universally rather than existentially quantified. Why is this the case? The type spine view represents all possible values of a datatype, therefore the existential variable must range over all possible types. This also explains the name of the constructor *AllEx*, which stands for all existential type representations.

There is another modification to the type spine view. The *Sig* constructor no longer carries constructor information. Instead, this information is stored at the top-level of the representation:

```
type TypeSpine a = [ConInfo (Signature a)]
```

This change is not strictly necessary but it is convenient. Suppose that the constructor information is still stored in *Sig*. Now, applications that need to perform a preprocessing pass using constructor information (for example, for more efficient parsing) would be forced to apply the function in *AllEx* only to obtain the constructor information. Having this information at the top-level, rather than at the *Sig* constructor, avoids the trouble of dealing with *AllEx* unnecessarily.

The function *typeSpine* has to be modified to deal with the new representation:

```
typeSpine :: Type a → TypeSpine a
typeSpine Int          = [conint i     (Sig i) | i ← [minBound..maxBound]]
typeSpine (Maybe a)    = [connothing (Sig Nothing)
                         ,conjust    (Sig Just    :⊕: a)]
typeSpine (Either a b) = [conleft    (Sig Left    :⊕: a)
                         ,conright   (Sig Right   :⊕: b)]
typeSpine (List a)     = [connil     (Sig [])
                         ,concons    (Sig (:)     :⊕: a :⊕: List a)]
typeSpine Dynamic      = [condynval  (AllEx (λa → Sig (DynVal a) :⊕: a))]
```

Now let us rewrite the *read* function using the new type spine view. First of all, the constructor is parsed in *read*, because the constructor information is now at the top-level:

```
read :: Type a → Parser a
read Int = readInt
read rep = alternatives [ readParen (conParser conrep)
                        | conrep ← typeSpine rep]
  where conParser conrep = token (conName conrep) ≫ gread (conVal conrep)
```

The function that performs generic parsing is not very different for the first two Signature constructors:

```
gread :: Signature a → Parser a
gread (Sig c)        = return c
gread (con :⊕: arg) = gread con `ap` read arg
```

The existential case is the most interesting one. We first parse the type representation, and then we continue with parsing the remaining part of the constructor.

```
gread (AllEx f) = readType ≫= applyBType (gread ∘ f)
```

This example effectively captures the *read* examples for dynamically typed values and for expression applications. The type representation is used to build the parser for

the remaining constructor arguments. This dependency is expressed using the bind operation on parsers (>>=). This means that the definition of generic read for existential types must be based on monadic parser combinators, and therefore applicative parser combinators cannot be used.

There is one function that we use to read type representations:

$$readType :: \mathsf{Parser\ BType}$$

Because type representations are somewhat special, we deal with them separately in Section 5.3.4.

5.3.2 Choice in the representation of existentials

There a choice in the representation of existential quantification. Consider the representation of *DynVal* given above. The function argument of *AllEx* receives a type representation and uses it to build the partially applied constructor value $Sig\ (DynVal\ a)$. This value requires only one more argument which is represented by a.

An alternative way to encode *DynVal* is to make all of the constructor arguments explicit:

$$\begin{aligned} &typeSpine :: \mathsf{Type\ a} \to \mathsf{TypeSpine\ a} \\ &\ldots \\ &typeSpine\ Dynamic = [condynval\ (AllEx\ (\lambda a \to Sig\ DynVal :\oplus: Type\ a :\oplus: a))] \end{aligned}$$

The two approaches differ in whether a generic function has access to the type representation in the application case ($:\oplus:$). It would seem that the second representation of *DynVal* is more flexible because it would allow the production of values different than a for the first argument. However, Type is a singleton type, so the only value (excluding $\perp$) that inhabits the type represented by *Type a* is a itself. It follows that the second representation of *DynVal* is not an improvement over the first. For this reason, we always choose to expose the representation of an existential by means of the existential case only (*AllEx*).

5.3.3 The existential case for consumer functions

Producer functions need a modified type spine view (TypeSpine) to handle existential types. Do we need to modify the spine view (Spine) for consumers too? After all, we were able to define *toSpine* for Dynamic using the existing view. There is a good reason why we still need to modify the spine view to handle existentials in an appropriate way. Consider the *read* and *show* functions for example. There is a clear dependency on the representation of existential types during parsing. It is not possible (or at least very impractical) to parse a dynamic value without first having the type representation for it. Therefore, existential type representations should appear earlier than the constructor arguments that depend on it in the text input used for parsing. This means that *show* must pretty print the type representation for the existential before the dependent

constructor arguments. However, the current spine view makes this difficult because the representation for the existential may appear in any position.

We solve the problem above making the dependence between existential types and constructor arguments explicit. Like the type spine view, the new constructor encodes the dependency on existentials using a function. The type variable is existentially quantified because in this case we are representing a specific constructor value:

```
data Spine :: * → * where
  Con  :: a → Spine a
  (:◇:) :: Spine (a → b) → Typed a → Spine b
  Ex   :: Type a → (Type a → Spine b) → Spine b
```

As in the type spine view, the constructor information is lifted out of the *Con* constructor onto the top-level. The new function *toSpine* is as follows:

```
toSpine :: Type a → a → ConInfo (Spine a)
toSpine Int          x          = conint x      (Con x)
toSpine (Maybe a)    (Just x)   = conjust       (Con Just    :◇: x :▷: a)
toSpine (Maybe a)    Nothing    = connothing    (Con Nothing)
toSpine (Either a b) (Left x)   = conleft       (Con Left    :◇: x :▷: a)
toSpine (Either a b) (Right y)  = conright      (Con Right   :◇: y :▷: b)
toSpine (List a)     (x : xs)   = concons       (Con (:)     :◇: x :▷: a
                                                             :◇: xs :▷: List a)
toSpine (List a)     []         = connil        (Con [])
toSpine Dynamic      (DynVal a x) = condynval   (Ex a dynSig)
  where dynSig a = Con (DynVal a) :◇: x :▷: a
```

The *show* function is modified as follows to use the constructor information that appears at the top-level:

```
show :: Type a → a → String
show rep x = paren (conName spinecon • gshow (conVal spinecon))
  where
    spinecon = toSpine rep x
```

Generic show does not change much for the two first spine cases:

```
gshow :: Spine a → String
gshow (Con con)     = ""
gshow (con :◇: arg) = gshow con • show (rep arg) (val arg)
```

For the existential case, generic show prints the type representation first and continues printing the remaining constructor values:

```
gshow (Ex a f) = showType a • gshow (f a)
```

The function for printing type representations is explained next:

```
showType :: Type a → String
```

5.3.4 Handling type representations

In the example above, we have used the function *readType* to parse a type representation. The function *readType* returns a boxed representation since the represented type is dynamically determined during parsing. Unfortunately, it is not easy to define producers that return boxed representations using generic programming. If special care is not taken, such functions may loop when invoked. In the following we describe the problem in more detail and we propose a solution.

Parsing type representations

The obvious way to parse a type representation is to do it generically by using the *read* function. To this end, we use generic read to parse boxed type representations:

```
readType :: Parser BType
readType = read BType
```

Unfortunately, the function given above is non-terminating. First, remember that BType uses existential quantification, and hence its type spine is:

$$typeSpine\ BType = [conboxed\ (AllEx\ (\lambda a \rightarrow Sig\ (Boxed\ a)))]$$

Since the type spine uses an existential case, *gread* would try to parse a BType-value calling *readType* recursively. Therefore, trying to parse a boxed type representation would lead to parsing an existential type, which leads to parsing a boxed type representation and so on.

How can we solve this problem? A desperate solution would be to give up using generic programming in the definition of *readType*. This approach is undesirable because every generic producer would need to have a type representation case. Worse even, every such case would have to handle all type representation constructors. If there are n generic functions and m represented types, the programmer would need to write $n \times m$ cases. Despite this significant problem, it is worth exploring a non-generic variant of *readType* and try to generalize it.

```
readType = alternatives (map readParen [ do token "Int"
                                           return (Boxed Int)
                                       , do token "Maybe"
                                           Boxed arg ← readType
                                           return (Boxed (Maybe arg))
                                       , do token "Either"
                                           Boxed left  ← readType
                                           Boxed right ← readType
                                           return (Boxed (Either left right))
                                       , do token "List"
                                           Boxed arg ← readType
                                           return (Boxed (List arg))
                                       ])
```

This example shows that parsing a type representation is no different than parsing a normal datatype in that the type argument of the GADT plays no role here. This example also illustrates the verbosity of writing such boilerplate without using generic programming.

The code of *readType* suggests that we could forget the "GADT-ness" of type representations during parsing. This is the first step we take towards being able to define generic producers for boxed representations, namely, defining the datatype of type codes, a non-GADT companion to type representations:

```
data TCode :: ∗ where
  CInt     :: TCode
  CMaybe   :: TCode → TCode
  CEither  :: TCode → TCode → TCode
  CList    :: TCode → TCode
  CArrow   :: TCode → TCode → TCode
  CType    :: TCode → TCode
  CDynamic :: TCode
  CTCode   :: TCode
```

Besides naming and the absence of a type argument, this datatype is identical to type representations. To make the relation between type codes and type representations precise, we introduce two conversion functions. The first function converts a type representation to a type code, erasing the type information in the process:

```
eraseType :: Type a → TCode
eraseType Int          = CInt
eraseType (Maybe a)    = CMaybe  (eraseType a)
eraseType (Either a b) = CEither (eraseType a) (eraseType b)
eraseType (List a)     = CList   (eraseType a)
eraseType (a :→ b)     = CArrow  (eraseType a) (eraseType b)
eraseType (Type a)     = CType   (eraseType a)
eraseType Dynamic      = CDynamic
eraseType TCode        = CTCode
```

Conversely, we want to be able to convert from a type code to a type representation. Note, however, that the resulting type-index depends on the value of the type code and hence the result is a boxed representation:

```
interpretTCode :: TCode → BType
interpretTCode CInt          = Boxed Int
interpretTCode (CMaybe a)    = applyTCode (Boxed ∘ Maybe) a
interpretTCode (CList a)     = applyTCode (Boxed ∘ List) a
interpretTCode (CType a)     = applyTCode (Boxed ∘ Type) a
interpretTCode CDynamic      = Boxed Dynamic
interpretTCode CTCode        = Boxed TCode
interpretTCode (CEither a b)
  = applyTCode (λr → applyTCode (Boxed ∘ Either r) b) a
interpretTCode (CArrow a b)
  = applyTCode (λr → applyTCode (Boxed ∘ (r :→ )) b) a


applyTCode :: ∀c.(∀a.Type a → c) → TCode → c
applyTCode f code = applyBType f (interpretTCode code)
```

Using type codes it is now possible to implement parsing of type representations generically. To implement *readType*, we parse a type code value and then we interpret it to obtain a type representation:

```
readType :: Parser BType
readType = read TCode >>= return ∘ interpretTCode
```

Here *TCode* is the type representation for type codes, we do not show the spine and type spine views for this datatype as they are no different from that of other datatypes.

Showing a type representation was no problem previously, we could have written *showType* as follows:

```
showType :: Type a → String
showType a = show (Type a) a
```

However, to remain compatible with *read* we use type codes as the means to pretty print type representations:

```
showType :: Type a → String
showType = show TCode ∘ eraseType
```

Summarizing, *readType* is a special function. It cannot be defined by instantiating *read* to boxed representations. Such an instantiation leads to non-termination because parsing a boxed representation uses the existential case of generic parsing, which in turn makes the recursive call to *readType*. To solve this problem, we defined type codes, a non-GADT analogue of type representations. Non-termination is no longer an issue with type codes. To parse a type code we no longer need to parse existential types, which prevents the recursive call to *readType*. This machinery enables the definition of *readType* as a generic program. This machinery can be reused for other generic producers, for example, see the definition of *enumerateType* in Section 5.4.

5.3.5 Equality of type representations

In this section we have introduced machinery to handle type representations generically, namely type codes and conversion functions between type codes and type representations. In Section 5.4, we show an advanced GADT example that requires a last piece of machinery: equality on type representations.

Below we show a function which compares two type representations, if the two representations are equal, it returns a proof that the two values represent the same type. First, we introduce the type of type equalities:

```
data TEq :: * → * → * where
  Refl :: TEq a a
```

A value of type TEq a b can be used to convince the type checker that two types a and b are the same at compile time. Since two type representations may not be the same, function *teq* returns the result in a monad:

```
teq :: Monad m ⇒ Type a → Type b → m (TEq a b)
teq Int            Int            = return Refl
teq (Maybe a)      (Maybe b)      = liftM   cong1 (teq a b)
teq (List a)       (List b)       = liftM   cong1 (teq a b)
teq (Either a c)   (Either b d)   = liftM2 cong2 (teq a b) (teq c d)
teq (Lam a c)      (Lam b d)      = liftM2 cong2 (teq a b) (teq c d)
teq (a :→ c)       (b :→ d)       = liftM2 cong2 (teq a b) (teq c d)
teq _              _              = fail "Different representations"
```

Here, we use *liftM* and *liftM2* to turn congruence functions into functions on monads. Congruence functions are used to lift equality proofs of types to arbitrary type constructors. These are defined as follows:

```
cong1 :: TEq a b → TEq (f a) (f b)
cong1  Refl      = Refl
cong2 :: TEq a b → TEq c d → TEq (f a c) (f b d)
cong2  Refl      Refl      = Refl
```

5.3.6 Type codes and dependently typed programming

In the literature of generic programming based on dependent types, sets of types having common structure are modelled by universes. Values known as universe codes describe type structure and an interpretation function makes the relationship between codes and types explicit.

The generic programming approach that this chapter describes would greatly benefit from the use of dependent types. Our approach is slightly redundant due to the necessity of both type representations and type codes. If we were to revise our approach to

use dependent types, the generic machinery would be based on type codes only. Previously, the type representation datatype described the relationship between types and the values that represent them. Using dependent types, this relationship would be defined by interpretation on codes and therefore type representations would not be necessary. Furthermore, producers like *readType* would no longer need to generate type representations. It follows that it would not be possible to accidentally define a non-terminating variant of such producers.

5.3.7 On the partiality of parsing typed syntax trees

Parsing is necessarily a partial operation. A parser for lists will fail to produce a value if the string to parse is not the textual representation of a list. Generic read is also a partial operation: the constructor names to be recognized in the input depend on the type representation argument of *gread*.

Generalized algebraic datatypes make the behaviour of *gread* more interesting. When a GADT is used, the set of constructors to recognize in the input will, in general, be a subset of all constructors in the GADT. For example, *gread* (*Expr Int*) parses all constructors with target type Expr Int but it fails to recognize the constructors *EFalse* and *ENot*. Note that this behaviour is closely related to type-checking: what would be type-checking errors in a different context are presented here as parsing errors.

A more interesting case is that of expression application. In this case, a representation for the function argument type is parsed and it steers the parsing of the function and the argument expressions. In this case, a type incompatibility between function and argument would be revealed as a parsing error.

5.4 Application: enumeration applied to simply typed lambda calculus

Generalized algebraic datatypes can encode sophisticated invariants using type-level constraints. We can combine such precise datatypes with generic producer functions, to generate values that have interesting properties. The example of this section combines a datatype representing terms of the simply typed lambda calculus with a generic function that enumerates all the values of a datatype. Using this function we can, for example, generate the terms that have the type of function composition.

5.4.1 Representing the simply typed lambda calculus

Terms of the simply typed lambda calculus can be represented as follows:

```
data Lam :: * → * → * where
  Vz  :: Lam a (EnvCons a e)
  Vs  :: Lam a e → Lam a (EnvCons b e)
  Abs :: Lam b (EnvCons a e) → Lam (a → b) e
  App :: Type a → Lam (a → b) e → Lam a e → Lam b e
```

The datatype Lam can be read as the typing relation for the simply typed lambda calculus. A value of type Lam a e represents the typing derivation for a term of type a in an environment e. Environments are encoded by list-like type constructors:

```
data EnvCons a e
data EnvNil
```

Each Lam constructor is a rule of the typing relation. The first constructor (*Vz*) represents a variable occurrence of type a, which refers to the first position of the environment (EnvCons a e). We can build a variable occurrence that refers to a deeper environment position by means of the weakening constructor *Vs*. Lambda abstractions are typed by means of the *Abs* constructor. In this case, a b-expression that is typeable in an environment containing a in the first position can be turned into a lambda abstraction of type a → b. The application constructor is almost like application in our previous example, *EApp*, except that *App* includes a representation for the existential type.

The spine representation for this datatype can be defined as follows:

```
toSpine (Lam a e)                 Vz          = convz  (Con Vz)
toSpine (Lam a (EnvCons b e)) (Vs tm)  = convs  (Con Vs  :◇: tm :▷: Lam a e)
toSpine (Lam (a :→ b) e)          (Abs tm) = conabs (Con Abs :◇: body)
  where body = tm :▷: Lam b (EnvCons a e)
toSpine (Lam b e)                 (App a tm1 tm2) = conapp (Ex a app)
  where app a = Con (App a) :◇: tm1 :▷: Lam (a :→ b) e :◇: tm2 :▷: Lam a e
```

The type representations are pattern matched in the *Vs* and *Abs* constructors to build the representation in the right hand side. The *App* constructor has an existential type, therefore we use *Ex* in the spine representation. Using the type representation, we can now print lambda terms.

For producer functions, we define the type spine view on Lam as follows:

```
typeSpine (Lam a e)
  = concat [[convz  (Sig Vz) | EnvCons a' e' ← [e], Refl ← teq a a']
           ,[convs  (Sig Vs :⊕: Lam a e') | EnvCons b e' ← [e]]
           ,[conabs (Sig Abs :⊕: Lam b (EnvCons a' e')) | a' :→ b ← [a]]
           ,[conapp (AllEx (λb → Sig (App b) :⊕: Lam (b :→ a) e
                                             :⊕: Lam b e))]
           ]
```

We test whether a constructor signature has the desired target type by performing pattern matching on type representations. The cases *Vz* and *Vs* are only usable if the

environment type argument is not empty. Additionally, the target type of *Vz* requires the equality of the type and the first position in the environment. Therefore, the *Vz* case invokes type equality (*teq*) on the term type (*a*) and the type of the first environment position (*a'*). The abstraction constructor (*Abs*) requires an arrow type, which is checked by pattern matching against an arrow type representation. The application constructor can always be used, because there is no restriction on the target type of *App*.

This type spine representation is more informative and larger than previous examples. The reason is that the GADT type argument is more complex because of the use of type-level environments. Furthermore, the type constraint in *Vz* requires the use of type equality (*teq*). Fortunately, it is possible to generate the type spine representation by induction on the syntax of the datatype declaration. It would be possible to automate this process using external tools such as DrIFT and Template Haskell if these tools supported GADTs.

5.4.2 Breadth first search combinators

The generic enumeration function generates all possible values of a datatype in breadth first search (BFS) order. The order used in the search corresponds to the search cost of terms generated. The type BFS is used for the results of a breadth first search procedure:

$$\textbf{type}\ \mathsf{BFS\ a} = [[\mathsf{a}]]$$

The type BFS represents a list of multisets sorted by cost. The first multiset contains terms of cost zero, the second contains terms of cost one and so on. Using this datatype, a consumer can inspect the terms up to a certain cost bound and hence the search does not continue if further terms are not demanded. This is useful because the enumeration function returns a potentially infinite list of multisets.

Multiple BFS values can be zipped together by concatenating multisets having terms of equal cost:

$$\begin{aligned}
&zip_{bfs} :: [\mathsf{BFS\ a}] \to \mathsf{BFS\ a} \\
&zip_{bfs}\ [\,] \ \ = [\,] \\
&zip_{bfs}\ xss = \textbf{if}\ all\ null\ xss\ \textbf{then}\ [\,]\ \textbf{else} \\
&\qquad\qquad concatMap\ head\ xss' : zip_{bfs}\ (map\ tail\ xss') \\
&\quad \textbf{where}\ xss' = filter\ (\neg \circ null)\ xss
\end{aligned}$$

It is more convenient to manipulate BFS results using monadic notation. Therefore, we define return and bind on BFS:

$$\begin{aligned}
&return_{bfs}\ x = [[x]] \\
&(\gg\!=_{bfs}) :: \forall \mathsf{a\ b.BFS\ a} \to (\mathsf{a} \to \mathsf{BFS\ b}) \to \mathsf{BFS\ b} \\
&(\gg\!=_{bfs})\ xss\ f = foldr\ (\lambda xs\ xss \to zip_{bfs}\ (map\ f\ xs +\!\!+ [[\,] : xss]))\ [\,]\ xss
\end{aligned}$$

Return creates a search result that contains a value of cost zero. Bind feeds the terms found in a search *xss* to a search procedure *f*. The cost of the term passed to *f* is added

to the costs of that search procedure. Consider, for example, the search results *xss* consisting of the terms $\lambda x\, y \to y$ and $\lambda x\, y \to x$ with costs three and four respectively; and a search procedure that produces a term of cost one by adding an abstraction to its argument:

$$\begin{aligned} xss &= [[\,],[\,],[\,],[Abs\ (Abs\ Vz)],[Abs\ (Abs\ (Vs\ Vz))]] \\ f\quad tm &= [[\,],[Abs\ tm]] \end{aligned}$$

Then, the expression $(xss \gg\!=_{bfs} f)$ evaluates to the following:

$$[[\,],[\,],[\,],[\,],[Abs\ (Abs\ (Abs\ Vz))],[Abs\ (Abs\ (Abs\ (Vs\ Vz)))]]$$

The two terms in the initial search result now have an additional abstraction argument and have costs of four and five respectively.

The cost addition property of bind can be stated more formally as follows:

$$\begin{aligned} &propBind :: \mathsf{BFS\ a} \to (\mathsf{a} \to \mathsf{BFS\ b}) \to \mathsf{Bool} \\ &propBind\ xss\ f = all\ (all\ costBind)\ (costs\ xss\ f) \\ &\quad \textbf{where}\ costBind\ (c,(c_{xss},c_f)) = c \equiv c_{xss} + c_f \\ &costs :: \mathsf{BFS\ a} \to (\mathsf{a} \to \mathsf{BFS\ b}) \to \mathsf{BFS}\ (\mathsf{Int},(\mathsf{Int},\mathsf{Int})) \\ &costs\ xss\ f = cost\ (cost\ xss \gg\!=_{bfs} \lambda(c_{xss},x) \to \\ &\qquad\qquad cost\ (f\,x) \gg\!=_{bfs} \lambda(c_f,y) \to \\ &\qquad\qquad return_{bfs}\ (c_{xss},c_f)) \end{aligned}$$

where *cost* annotates each BFS result value with its cost:

$$\begin{aligned} &cost :: \mathsf{BFS\ a} \to \mathsf{BFS}\ (\mathsf{Int},\mathsf{a}) \\ &cost = zipWith\ (\lambda sz \to map\ ((,)\ sz))\ [0..] \end{aligned}$$

Additionally we use a function that increases the cost of the values found in a search procedure:

$$\begin{aligned} &spend :: \mathsf{Int} \to \mathsf{BFS\ a} \to \mathsf{BFS\ a} \\ &spend\ n = (!!n) \circ iterate\ ([\,]:) \end{aligned}$$

When using a very expensive search procedure, it is useful to increase the cost of terms exponentially:

$$\begin{aligned} &raise :: \mathsf{Int} \to \mathsf{BFS\ a} \to \mathsf{BFS\ a} \\ &raise\ base\ xss = traverse\ 0\ xss \\ &\quad \textbf{where}\ traverse\ \_\quad [\,] \qquad = [\,] \\ &\qquad\qquad traverse\ 0\quad (xs:xss) = [\,]:xs:traverse\ 1\ xss \\ &\qquad\qquad traverse\ exp\ (xs:xss) = spend\ (base^{exp} - base^{exp-1} - 1) \\ &\qquad\qquad\qquad\qquad\qquad\qquad (xs:traverse\ (exp+1)\ xss) \end{aligned}$$

For example, *spend* 2 *xss* and *raise* 2 *xss* evaluate to:

$$[[],[],[],[],[],[Abs\ (Abs\ Vz)],[Abs\ (Abs\ (Vs\ Vz))]]$$

and

$$[[],[],[],[],[],[],[],[],[Abs\ (Abs\ Vz)],\\ [],[],[],[],[],[],[],[Abs\ (Abs\ (Vs\ Vz))]]$$

respectively.

5.4.3 Generic enumeration

The generic enumeration function returns values of a datatype, classified by cost in increasing order:

$$\begin{aligned}&enumerate :: \mathsf{Type\ a} \rightarrow \mathsf{BFS\ a}\\ &enumerate\ a = zip_{bfs}\ [genumerate\ (conVal\ s) \mid s \leftarrow typeSpine\ a]\end{aligned}$$

At the top-level, function *genumerate* is invoked on each constructor signature and the resulting search results are zipped together.

The first case of *genumerate* returns the constructor value as the search result assigning it a cost of one. The second case performs search recursively on the function and argument parts and combines the results using BFS monadic application.

$$\begin{aligned}&genumerate :: \mathsf{Signature\ a} \rightarrow \mathsf{BFS\ a}\\ &genumerate\ \ (Sig\ c) = spend\ 1\ (return_{bfs}\ c)\\ &genumerate\ \ (fun :\oplus: arg) = genumerate\ fun\ `ap_{bfs}`\ enumerate\ arg\end{aligned}$$

The third case deals with existential types and hence in our particular application it deals with expression application. This case first enumerates all possible types, then, for each type, a constructor signature is constructed using f and enumeration is called recursively:

$$genumerate\ (AllEx\ f) = enumerateType \gg\!=_{bfs} genumerate \circ applyBType\ f$$

As usual with producer functions, *enumerateType* returns a boxed representation. The enumeration of types is performed on type codes, which are converted to boxed representations using *interpretTCode*.

$$\begin{aligned}&enumerateType :: \mathsf{BFS\ BType}\\ &enumerateType = raise\ 4\ (enumerate\ TCode) \gg\!=_{bfs} return_{bfs} \circ interpretTCode\end{aligned}$$

For the examples in this chapter, we are not interested in values that have very complex existential types. Therefore, we keep their size small by assigning an exponential cost to existentials. This also has the effect of reducing the search space, which makes the generation of interesting terms within small cost upperbounds more likely.

5.4.4 Term enumeration in action

For convenience, we define a wrapper function to perform enumeration of lambda terms:

```
enumerateLam :: Type a → Int → BFS (Lam a EnvNil)
enumerateLam a cost = take (cost + 1) (enumerate (Lam a EnvNil))
```

Our term datatype can perfectly deal with open terms. But the user interface becomes simpler if only closed terms are provided. Therefore, the wrapper function only generates closed lambda terms.

A direct invocation of the enumeration function will result in an attempt to generate an infinite number of terms. For convenience, our wrapper function takes a cost upperbound that limits the cost of terms that are reported. Because of lazy evaluation, the search procedure stops when all terms within the cost bound are reported. The user may choose to increase the cost upperbound in subsequent invocations if the desired term is not found.

The language that the Lam datatype represents is very simple. There are no datatypes, recursion, and arithmetic operations. For example, we cannot expect the enumeration function to generate the successor or predecessor functions for naturals if functions of the type Int → Int are requested. In principle, it is not difficult to extend the language by adding the appropriate constants to Lam. For example, we could add naturals and arithmetic operations on them. We could also add list constructors and elimination functions and even recursion operators such as catamorphisms and paramorphisms.

However, we can keep our language simple and still generate many interesting terms. We focus our attention to parametrically polymorphic functions. Although we do not model parametric polymorphism explicitly in Lam, such functions are naturally generated when the requested type is an instance of the polymorphic type. For instance, a request with type Int → Int generates the identity function.

To make the intent of generating polymorphic functions more explicit, we define a few types that are uninhabited in the Lam language. These types play the role of type variables in polymorphic type signatures:

```
data A
data B
data C
data D
```

Of course, we also introduce the corresponding type representation constructors:

```
data Type :: ∗ → ∗ where
  ...
  A :: Type A
  B :: Type B
  C :: Type C
  D :: Type D
```

In our first example, we generate the code for the identity function. The type of the identity function is $\forall a.a \to a$, which in our notation translates to $A :\to A$. The function we expect to generate is $\lambda x \to x$, which in Lam is written as *Abs Vz*. This term consists of two constructors, therefore a cost upperbound of two should suffice to generate it. The application *enumerateLam* $(A :\to A)$ 2 results in:

$$[[\,],[\,],[\mathit{Abs}\ \mathit{Vz}]]$$

It is instructive to sketch the search procedure as it looks for the identity function. First, *enumerate* is called on the identity type with a closed environment. This function calls *genumerate* on all constructor signatures that match the desired type. The two variable cases *Vz* and *Vs* are not considered, because they cannot be used with an empty environment. Application can always be used but recall that it requires an existential type representation, so the cost is at least 5, which is more expensive than the function that we are looking for. The abstraction case matches the identity type so enumeration is called recursively to generate the abstraction body. Now, a term of type A is requested with a type A in the first position in the environment. The case *Vz* matches perfectly with this request so the term *Abs Vz* is returned with cost two.

There are infinitely many lambda calculus terms of a given type when that type is inhabited. A simple way to way to obtain a new term is by creating a redex that reduces to the term that we currently have. For example, we can obtain a new identity function by adding a redex in the function body: $\lambda x \to (\lambda x \to x)\ x$. Can this term be found by our enumeration function? Yes, provided that we increase the cost upperbound to include that of our new term. The new term is essentially two identity functions plus an application constructor, which makes a cost upperbound of nine. We evaluate *enumerateLam* $(A :\to A)$ 9 which yields:

$$[[\,],[\,],[\mathit{Abs}\ \mathit{Vz}],[\,],[\,],[\,],[\,],[\,],[\,],[\mathit{Abs}\ (\mathit{App}\ A\ (\mathit{Abs}\ \mathit{Vz})\ \mathit{Vz})]]$$

This example shows that the search space is somewhat redundant. A way to speed up term search would be to avoid the generation of redundant terms by adding constraints to Lam. For example, we could avoid redexes by preventing the generation of abstractions in the left part of applications.

Our next example is the generation of the application function. This function has type $\forall \mathsf{a}\ \mathsf{b}.(\mathsf{a} \to \mathsf{b}) \to \mathsf{a} \to \mathsf{b}$, which in our notation is written $((A :\to B) :\to A :\to B)$. To generate the application functions we evaluate *enumerateLam* $((A :\to B) :\to A :\to B)$ 10 which results in:

$$[[\,],[\,],[\mathit{Abs}\ \mathit{Vz}],[\,],[\,],[\,],[\,],[\,],[\,],[\,],[\mathit{Abs}\ (\mathit{Abs}\ (\mathit{App}\ A\ (\mathit{Vs}\ \mathit{Vz})\ \mathit{Vz}))]]$$

These are the encodings for the functions $\lambda x \to x$ and $\lambda x\, y \to x\, y$.

Our last example is function composition. The type of this function is $\forall \mathsf{a}\ \mathsf{b}\ \mathsf{c}.(\mathsf{b} \to \mathsf{c}) \to (\mathsf{a} \to \mathsf{b}) \to \mathsf{a} \to \mathsf{c}$. To generate composition, we evaluate

$$\mathit{last}\ (\mathit{enumerateLam}\ ((B :\to C) :\to (A :\to B) :\to (A :\to C))\ 19)$$

which yields to the encoding of $\lambda x\, y\, z \to x\ (y\ z)$:

$$[Abs\ (Abs\ (Abs\ (App\ B\ (Vs\ (Vs\ Vz))\ (App\ A\ (Vs\ Vz)\ Vz))))]$$

5.5 Related work

To the best of our knowledge, only the spine approach (Hinze et al. 2006; Hinze and Löh 2006) enables generic programming on generalized algebraic datatypes in Haskell. This is the approach on which the work in this chapter is based. Because both the spine and the type spine view can encode GADTs, both consumer and producer functions can be defined on such datatypes. Interestingly, to properly support GADTs for producer functions, the approach should also support existential types. For example, when reading a GADT from disk, we may want the GADT type argument to be dynamically determined from the disk contents. Therefore, we would existentially quantify over that argument. However, the spine approach supports existential types for consumers but not for producers.

Generalized algebraic datatypes are inspired by inductive families in the dependent types community. We are aware of two approaches (Benke et al. 2003; Morris 2007) that support definitions by induction on the structure of inductive families. Both approaches make essential use of evaluation on the type level to express the constraints over inductive families. Examples of type families on which generic programming is applied include trees (indexed by their lower and upper size bounds), finite sets, vectors and telescopes. Neither approach gives examples for the support of existential types so it is not clear whether these are supported.

Weirich (2002) proposes a language that provides a construct to perform runtime case analysis on types. In order to support universal and existential quantification, the language includes analyzable type constants for both quantifiers. This approach supports the definition of consumers and producers. Moreover if the language is extended with polymorphic kinds it supports quantification over arbitrarily kinded types.

Koopman and Plasmeijer (2007) generate lambda calculus terms by performing systematic enumeration based on a grammar. To reduce the size of the search space, the grammar has syntactic restrictions such as the applications of certain operands are always saturated, and recursive calls are always guarded by a conditional. The candidate terms are then reported back to the user based on whether they satisfy an input-output specification, which is established by evaluation. The work of Katayama (2005) generates lambda calculus terms based on syntax and uses a type-checking phase to discard ill-typed terms. The candidate terms are then evaluated and checked against an input-output specification.

Djinn (Augustsson 2005) generates lambda calculus terms based on a user-supplied type. This tool implements the decision procedure for intuitionistic propositional calculus due to Dyckhoff (1992). As in our approach, Djinn generates only well-typed terms so there is no need for a type checking phase to discard ill-typed terms.

The main difference between the work of Koopman and Plasmeijer (2007), Katayama (2005), and ours is that our generator is typed-based. It follows that our generator never returns ill-typed terms because the search space is reduced by means of type-level constraints in the GADT. Generating ill-typed terms has advantages. For example, Koopman's approach can generate the Y-combinator. On the other hand, ill-typed terms are usually not desirable, so these have to be discarded through either evaluation (Koopman) or a type-checking phase (Katayama), which slows down the generation algorithm. In Koopman's work the generation of ill-typed terms is prevented to some extent by the syntactic constraints imposed on the grammar.

A type-based generator, such as Djinn and our approach, is able to synthesize polymorphic functions without the need for input-output specifications. The approach of Katayama can also be used in this way because of its type-checker. Koopman's work, however, cannot generate polymorphic functions based solely on type information.

Djinn supports user-defined datatypes. Katayama and Koopman's generators are able to generate recursive programs. Our approach currently generates programs for a rather spartan language. However, it should be possible to add introduction and elimination constants for (recursive) datatypes, and recursion operators such as catamorphisms and paramorphisms.

Both Djinn and our approach enumerate terms guided by type information. However, the two approaches are very different. Djinn has a carefully crafted algorithm that handles the application of functional values in such a way that it is not necessary to exhaustively enumerate the infinite search space. As a consequence, Djinn is able to detect that a type is uninhabited in finite time. In contrast, our approach produces function applications by means of exhaustive enumeration. First, all the possible types of an argument are enumerated, and, for each of them, function and argument terms are enumerated to construct an application. The good side of an exhaustive approach like ours is that it can generate all possible terms of a given type. For example, it can generate all Church numerals, whereas Djinn only generates those corresponding to zero and one. On the bad side, if unbounded, our approach does not terminate when trying to generate a term for an uninhabited type.

We have not performed a careful performance comparison but we believe that our generator may be the slowest of the approaches considered here. Probably the main culprit for inefficiency is the implementation of the existential case. Currently this case enumerates all possible types, even if no applications for that argument type can be constructed. Ill-typed terms are never generated, but resources are nevertheless consumed when attempting to enumerate terms having possibly uninhabited types. It is difficult to make the algorithm smarter about generating types because, being generic, it does not make assumptions about the particularities of lambda calculus. On the other hand, it is possible to reduce the search space by adjusting the definition of `Lam`. For example, we could forbid the formation of redexes to avoid redundancy of terms, or even adopt the syntactic restrictions used in Koopman's work.

While our approach may be less efficient, it has the virtue of simplicity: the core of the generation algorithm consists of roughly a dozen lines of code and there is no need for an evaluation or a type checking phase. Furthermore, it has the advantage

of an elegant separation between the grammar constraints and the formulation of the enumeration algorithm. This allows us to use the enumeration function to generate other languages, whereas the other generators are specific to lambda calculus.

5.6 Conclusions

We have presented an extension of the spine approach to generic programming, which supports the definition of generic producers for existential types. This extension allows the definition of, for example, generic read for datatypes that use existential quantification.

Our approach opens the way for a new application of generic programming. By taking the standard enumeration generic function and extending it with a case for existentials, we obtain a function that enumerates well-typed terms. For example, we can instantiate enumeration to the GADT that represents terms of the simply typed lambda calculus and use the resulting function to search for terms that have a given type. Such an application was not previously possible because producers that handle existential types could not be generically defined.

6 Generating Generic Functions

In this chapter we present an approach to the generation of generic functions from user-provided specifications. The specifications consist of the type of a generic function, examples of instances that it should "match" when specialized, and properties that the generic function should satisfy. We use the type-based function generator Djinn to generate terms for the generic function base cases. Candidate generic functions are then pruned by testing against properties, and by testing generic functions specializations against the provided examples. Using this approach we have been able to generate the generic equality, map, and zip functions.

6.1 Introduction

How do we introduce a datatype-generic function? Usually we do the following:

> Here is the instance of the function on the data type List, and here is the instance of the generic function on some kind of Tree. The pattern in these definitions should be clear. It follows that this is the type of the generic function, and the different lines in the definition of the generic function are as follows. This is the correct generic function, because if I instantiate it on the types List and Tree, I get functions with the same semantics as the functions I gave when I introduced the function.

Examples of this approach have been given by Hinze (2000b) for generalized folds, Gibbons (1998) for accumulations, Backhouse et al. (1999) for maps, and by many other authors introducing new generic functions. This approach has worked well, and both readers and students are usually convinced by the argument. However, it raises the question if there is any choice when we define a generic function after giving two or more examples on specific data types, and whether this procedure can be automated.

Why would we want to generate generic functions from examples?

- First, an algorithm that generates generic functions from examples (and possibly types and properties) is a convenient way to *specify* generic functions.
- Second, it offers (novice) users help when writing a generic function.
- Third, it formalizes the informal procedure with which we started this chapter.

Recently, there has been growing interest in the automatic generation of functions from user-provided specifications (Katayama 2005; Koopman and Plasmeijer 2006;

Augustsson 2005), mainly consisting of a type and a set of input-output examples. However, the function generation research above is not directly applicable to the generation of generic functions due to their different nature: the approach of Koopman and Plasmeijer (2006) does not seem to be able to generate higher-order functions, which is essential for generating generic functions, while Djinn (Augustsson 2005) and Katayama's MagicHaskeller (Katayama 2005) generate polymorphic functions but not generic functions.

In this chapter we propose a procedure for the generation of generic functions in Generic Haskell (Löh et al. 2008). Generic Haskell is an extension of Haskell that supports the definition of datatype-generic functions.

Generic functions are generated from a user-provided specification consisting of the type of a generic function, examples of instances, and properties that the generic function should satisfy. The generation procedure proceeds as follows:

- The cases of which a generic function consists are generated from the specialized generic-function type. In this chapter we use the Djinn tool (Augustsson 2005) for the type-based generation.

- The generated terms are pruned by testing each term against properties it should satisfy. We use the QuickCheck library (Claessen and Hughes 2000) for testing.

- The set of candidate generic functions is constructed by taking the cross-product of the generated function cases.

- This set is pruned by testing each candidate against examples instances. Candidate functions are instantiated using the specialization algorithm of Generic Haskell (Hinze and Jeuring 2003; Löh 2004).

This chapter is organized as follows. Section 6.2 introduces type-indexed and generic functions and briefly shows how Generic Haskell generates code for generic functions. Section 6.3 shows how a generic programmer typically writes a generic function in Generic Haskell. Section 6.4 presents the central ideas of this chapter: how do we generate a generic function given its type, example instances, and properties it should satisfy. It introduces type-based function generation and shows how Djinn is used to generate generic functions from types. Section 6.5 explains the design choices we made for our tool and briefly explains its implementation. Section 6.6 reports the results of our research. Section 6.7 describes related work, future work, and concludes.

6.2 Generic functions in Generic Haskell

In this section we briefly introduce type-indexed functions and generic functions. The material here and the remaining sections are strongly related to Generic Haskell. More details about Generic Haskell can be found in Löh (2004).

Type-indexed functions

A type-indexed function is a function that is defined on every type of a family of types. The members of a family are type expressions formed by a finite number of type constructors. For example, we consider the family formed by the unit (Unit), sum ($+$) and product ($\times$) types, which are defined as follows:

$$\begin{aligned}&\textbf{data}\ \mathsf{Unit} = \mathit{Unit}\\&\textbf{data}\ \mathsf{a}+\mathsf{b} = \mathit{Inl}\ \mathsf{a} \mid \mathit{Inr}\ \mathsf{b}\\&\textbf{data}\ \mathsf{a}\times\mathsf{b} = \mathsf{a}\times\mathsf{b}.\end{aligned}$$

For example, the types $\mathsf{Unit}+\mathsf{Unit}$ and $\mathsf{Unit}\times\mathsf{Unit}$ inhabit the family formed by the datatype definitions above.

The definitions of type-indexed functions are given by case analysis on types. For example, the definition of equality on unit, sum and product types is as follows:

$$\begin{aligned}&\mathit{equals}\{\!|\mathsf{a}::*|\!\}::(\mathit{equals}\{\!|\mathsf{a}|\!\})\Rightarrow \mathsf{a}\to\mathsf{a}\to\mathsf{Bool}\\&\mathit{equals}\{\!|\mathsf{Unit}|\!\}\ \ \mathit{Unit}\ \ \mathit{Unit}\ \ = \mathit{True}\\&\mathit{equals}\{\!|\alpha+\beta|\!\}\ (\mathit{Inl}\ x)\ \ (\mathit{Inl}\ y)\ \ = \mathit{equals}\{\!|\alpha|\!\}\ x\ y\\&\mathit{equals}\{\!|\alpha+\beta|\!\}\ (\mathit{Inr}\ x)\ \ (\mathit{Inr}\ y)\ \ = \mathit{equals}\{\!|\beta|\!\}\ x\ y\\&\mathit{equals}\{\!|\alpha+\beta|\!\}\ \_\ \ \_\ \ = \mathit{False}.\\&\mathit{equals}\{\!|\alpha\times\beta|\!\}\ (x_1\times y_1)\ (x_2\times y_2) = \mathit{equals}\{\!|\alpha|\!\}\ x_1\ x_2\wedge \mathit{equals}\{\!|\beta|\!\}\ y_1\ y_2\end{aligned}$$

This function definition uses Generic Haskell syntax. The signature has the familiar type of equality, with the addition that the type variable a generalizes over specific types. The braces denote the type whose values the function case handles. The cases for polymorphic types (sum and product) make a recursive call to equality to compare the constituent values.

The context $(\mathit{equals}\{\!|\mathsf{a}|\!\})\Rightarrow$ in this signature specifies that *equals* has a *dependency* (Löh et al. 2003) on *equals*. Informally, dependencies keep track of calls to other generic functions including the function being defined. Because the function calls itself in the case of polymorphic types, the signature includes the dependency constraint.

The Generic Haskell compiler can specialize the definitions of type-indexed functions to types that are in the type family. For example, consider examples of specialization types:

$$\begin{aligned}&\mathit{equals}\{\!|\mathsf{Unit}+\mathsf{Unit}|\!\}::\mathsf{Unit}+\mathsf{Unit}\to\mathsf{Unit}+\mathsf{Unit}\to\mathsf{Bool}\\&\mathit{equals}\{\!|\mathsf{Unit}\times\mathsf{Unit}|\!\}::\mathsf{Unit}\times\mathsf{Unit}\to\mathsf{Unit}\times\mathsf{Unit}\to\mathsf{Bool}.\end{aligned}$$

The specializations are generated by Generic Haskell by essentially performing an interpretation on expressions of the type language. For instance, it translates a type constant to its corresponding type-indexed definition and type application to expression application:

$$\mathit{equals}\{\!|\mathsf{Unit}+\mathsf{Unit}|\!\} = \mathit{equals}\{\!|+|\!\}\ \mathit{equals}\{\!|\mathsf{Unit}|\!\}\ \mathit{equals}\{\!|\mathsf{Unit}|\!\}$$

Here the sum case for equality takes equality arguments to compare the constituents of sum values. In the sum case definition, the $\mathit{equals}\{\!|\alpha|\!\}$ and $\mathit{equals}\{\!|\beta|\!\}$ values can be regarded as syntactic sugar that denotes the function arguments in the translated code.

Generic functions

The domain of a type-indexed function is limited to the type expressions that can be formed from the type-cases handled by the function. In contrast, a generic function can handle datatype values that do not belong to the domain of the type family. For example, using the definition for equality given above, Generic Haskell can generate equality on lists, which essentially behaves as the following hand-written version:

$$\begin{aligned}
&\mathit{equals}\{\!|[\alpha]|\!\}\ [\,] \quad\ [\,] \quad\ = \mathit{True}\\
&\mathit{equals}\{\!|[\alpha]|\!\}\ (x:xs)\ (y:ys) = \mathit{equals}\{\!|\alpha|\!\}\ x\ y \wedge \mathit{equals}\{\!|[\alpha]|\!\}\ xs\ ys\\
&\mathit{equals}\{\!|[\alpha]|\!\}\ _ \quad\ _ \quad\ = \mathit{False}.
\end{aligned}$$

To generate this instance of equality, Generic Haskell generates a type that represents the structure of lists. The structure type is used to specialize the generic function and subsequently an adaptor function turns the equality on the structure of lists to equality on lists:

$$\mathit{equals}\{\!|[\alpha]|\!\} = \mathit{adaptor}\ \mathit{equals}\{\!|[\alpha]^\circ|\!\}$$

In the following we give examples of structure types and describe the role these play in the specialization process.

Structure types

In Generic Haskell the structure of datatypes is described using unit, sum and product datatypes. If the datatype definitions contain primitive types such as integers and floats, the generic function is required to provide a case for these. Given datatype declarations such as

$$\begin{aligned}
&\textbf{data}\ [\mathsf{a}] \quad\ = [\,] \mid \mathsf{a} : [\mathsf{a}]\\
&\textbf{data}\ \mathsf{Tree\ a\ b} = \mathit{Tip}\ \mathsf{a} \mid \mathit{Node}\ (\mathsf{Tree\ a\ b})\ \mathsf{b}\ (\mathsf{Tree\ a\ b}),
\end{aligned}$$

the Generic Haskell compiler derives type synonyms that contain the structural representation of the declared datatypes:

$$\begin{aligned}
&\textbf{type}\ [\mathsf{a}]^\circ \quad\ = \mathsf{Unit} + \mathsf{a} \times \mathsf{List\ a}\\
&\textbf{type}\ \mathsf{Tree}^\circ\ \mathsf{a\ b} = \mathsf{a} + \mathsf{Tree\ a\ b} \times \mathsf{b} \times \mathsf{Tree\ a\ b}.
\end{aligned}$$

We assume that $\times$ binds stronger than $+$ and both type constructors associate to the right. Note that the representation of a recursive type is not recursive, and refers to the recursive type itself: the representation of a type in Generic Haskell only represents the structure of the top level of the type.

The specialization of equality to the structure of lists is:

$$\mathit{equals}\{\!|[\mathsf{a}]^\circ|\!\} = \mathit{equals}\{\!|+|\!\}\ \mathit{equals}\{\!|\mathsf{Unit}|\!\}\ (\mathit{equals}\{\!|\times|\!\}\ \mathit{equals}\{\!|\mathsf{a}|\!\}\ \mathit{equals}\{\!|[\mathsf{a}]|\!\}).$$

Embedding-projection pairs
Embedding-projection pairs are witnesses that a datatype and a structure type are isomorphic. The isomorphism witnesses are a pair of functions that convert a datatype value to a structure value and back.

$$\textbf{data}\ \mathsf{EP\ a\ b} = \mathit{EP}\{\mathit{from} :: \mathsf{a} \to \mathsf{b}, \mathit{to} :: \mathsf{b} \to \mathsf{a}\}.$$

The embedding projection for lists is automatically generated by Generic Haskell:

$$\begin{aligned}
&\mathit{conv}_{[]} :: \forall \mathsf{a}\,.\,\mathsf{EP}\ [\mathsf{a}]\ [\mathsf{a}]^{\circ}\\
&\mathit{conv}_{[]} = \mathit{EP}\ \mathit{from}_{[]}\ \mathit{to}_{[]}\\
&\mathit{from}_{[]} \qquad\qquad :: \forall \mathsf{a}\,.\,[\mathsf{a}] \to [\mathsf{a}]^{\circ}\\
&\mathit{from}_{[]}\ [\,] \qquad\quad = \mathit{Inl}\ \mathit{Unit}\\
&\mathit{from}_{[]}\ (x : xs) \qquad = \mathit{Inr}\ (x \times xs)\\
&\mathit{to}_{[]} \qquad\qquad :: \forall \mathsf{a}\,.\,[\mathsf{a}]^{\circ} \to [\mathsf{a}]\\
&\mathit{to}_{[]}\ (\mathit{Inl}\ \mathit{Unit}) \qquad = [\,]\\
&\mathit{to}_{[]}\ (\mathit{Inr}\ (x \times xs)) = x : xs.
\end{aligned}$$

Tying the knot
Using structural representation types and embedding-projection pairs, a call to a generic function on a data type T is reduced to a call on type T°. For example, for equality we obtain a function of type $[\mathsf{a}]^{\circ} \to [\mathsf{a}]^{\circ} \to \mathsf{Bool}$. To convert this function to a function of type $[\mathsf{a}] \to [\mathsf{a}] \to \mathsf{Bool}$, the Generic Haskell compiler generates an adaptor function, which is defined using a generic bidirectional mapping function (Hinze 2000a; Löh 2004), combined with the embedding projection for lists:

$$\begin{aligned}
&\mathit{bimap}\{\!|\mathsf{a} :: *, \mathsf{b} :: *|\!\} :: (\mathit{bimap}\{\!|\mathsf{a},\mathsf{b}|\!\}) \Rightarrow \mathsf{EP\ a\ b}\\
&\mathit{adaptor} :: \forall \mathsf{a}\,.\,([\mathsf{a}]^{\circ} \to [\mathsf{a}]^{\circ} \to \mathsf{Bool}) \to ([\mathsf{a}] \to [\mathsf{a}] \to \mathsf{Bool})\\
&\mathit{adaptor} = \mathit{to} \circ \mathit{bimap}\{\!|\to|\!\}\ \mathit{conv}_{[]}\ (\mathit{bimap}\{\!|\to|\!\}\ \mathit{conv}_{[]}\ \mathit{bimap}\{\!|\mathsf{Bool}|\!\})
\end{aligned}$$

6.3 Writing a generic function

This section describes the steps a generic programmer may follow to arrive at the definition of a generic function.

Suppose that we wish to define generic equality. The first step is to write down examples of specific instances to understand the general pattern of equality. Consider, for example, equality for lists and trees.

$$
\begin{aligned}
&\mathit{equals}_{[\mathsf{Int}]} :: [\mathsf{Int}] \to [\mathsf{Int}] \to \mathsf{Bool} \\
&\mathit{equals}_{[\mathsf{Int}]}\ [\,]\ \ [\,] = \mathit{True} \\
&\mathit{equals}_{[\mathsf{Int}]}\ (x : xs)\ (y : ys) = \mathit{equals}_{\mathsf{Int}}\ x\ y \wedge \mathit{equals}_{[\mathsf{Int}]}\ xs\ ys \\
&\mathit{equals}_{\mathsf{Tree}} :: (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to (\mathsf{b} \to \mathsf{b} \to \mathsf{Bool}) \to \mathsf{Tree\ a\ b} \to \mathsf{Tree\ a\ b} \to \mathsf{Bool} \\
&\mathit{equals}_{\mathsf{Tree}}\ \mathit{equals}_{\mathsf{a}}\ \mathit{equals}_{\mathsf{b}}\ (\mathit{Tip}\ x)\ (\mathit{Tip}\ y) = \mathit{equals}_{\mathsf{a}}\ x\ y \\
&\mathit{equals}_{\mathsf{Tree}}\ \mathit{equals}_{\mathsf{a}}\ \mathit{equals}_{\mathsf{b}}\ (\mathit{Node}\ x_1\ y_1\ z_1)\ (\mathit{Node}\ x_2\ y_2\ z_2) \\
&\quad = \mathit{equals}_{\mathsf{Tree}}\ \mathit{equals}_{\mathsf{a}}\ \mathit{equals}_{\mathsf{b}}\ x_1\ x_2 \wedge \\
&\qquad \mathit{equals}_{\mathsf{b}}\ \ y_1\ y_2 \wedge \\
&\qquad \mathit{equals}_{\mathsf{Tree}}\ \mathit{equals}_{\mathsf{a}}\ \mathit{equals}_{\mathsf{b}}\ z_1\ z_2,
\end{aligned}
$$

where $\mathit{equals}_{\mathsf{Int}}$ is an equality function on integers. Equality on polymorphic trees takes two additional functional arguments to compare the values of types a and b. The types of these two examples are subsumed by the type of the generic equality function:

$$\mathit{equals}\{\!|\mathsf{a} :: *|\!\} :: (\mathit{equals}\{\!|\mathsf{a}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}.$$

To specialize this type to that of equality on $[\mathsf{Int}]$ values, we substitute the variable a in the signature. The specialization to polymorphic trees is less obvious. The specialization procedure depends on the kind of the type so that the shape of signatures is different for polymorphic types. In particular, the specialization of this signature for polymorphic trees, yields the signature of a function that compares two tree values but additionally requires comparison functions for a and b values. In short, it yields the same signature of $\mathit{equals}_{\mathsf{Tree}}$. This kind-indexed specialization procedure was introduced by Hinze (2000c).

The type-specific definitions suggest that, in general, two values are equal if their constructors are the same and if equality holds for every pair of corresponding constructor fields. It is natural to use this insight to write the definition of generic equality in Generic Haskell, taking into account the way that data types are represented by structural representations. For instance, the data type $[\,]$ is represented in Generic Haskell by the structural representation **type** $[\mathsf{a}]^\circ = \mathsf{Unit} + \mathsf{a} \times [\mathsf{a}]$. Sums represent choice between constructors, so we encode the fact that equality only holds for matching constructors as follows:

$$
\begin{aligned}
&\mathit{equals}\{\!|\alpha + \beta|\!\}\ (\mathit{Inl}\ x)\ (\mathit{Inl}\ y) = \mathit{equals}\{\!|\alpha|\!\}\ x\ y \\
&\mathit{equals}\{\!|\alpha + \beta|\!\}\ (\mathit{Inr}\ x)\ (\mathit{Inr}\ y) = \mathit{equals}\{\!|\beta|\!\}\ x\ y \\
&\mathit{equals}\{\!|\alpha + \beta|\!\}\ \_\ \ \ \_ = \mathit{False}.
\end{aligned}
$$

If the values match the same alternative, the recursive call to *equals* compares the remaining structure, namely the constructor fields or the remaining constructor choices.

As products represent the fields of a constructor, we compute the equality of the constructor fields by comparing the corresponding components of the two products:

$$\mathit{equals}\{\!|\alpha \times \beta|\!\}\ (x_1 \times y_1)\ (x_2 \times y_2) = \mathit{equals}\{\!|\alpha|\!\}\ x_1\ x_2 \wedge \mathit{equals}\{\!|\beta|\!\}\ y_1\ y_2.$$

The remaining case for the unit data type handles constructors with no fields and is trivial:

$$equals\{|\mathsf{Unit}|\}\ Unit\ Unit = True.$$

Types play a prominent rôle in the definition of equality on sums and products. Neither case has specific information about the components of sums and products. Therefore, the only possibility is to test for equality by recursively calling the generic equality function on the component types.

We summarize the informal procedure by which we obtain the generic equality function. First, we look at some instances of equality to gain insight into the general pattern of equality. Second, we obtain the type of the generic function that subsumes the types of the examples. Third, we write the definition taking into account the structure type encoding of data types and using the type information available in each case.

6.4 Generating generic functions

This section outlines our approach to generating generic functions, which is inspired by the informal procedure for writing generic functions described in the previous section.

To generate a generic function, a user specifies the type of the desired generic function, a set of instance examples for the desired function, and the properties that the function should satisfy. For example, the user might provide the type signature for generic equality given in the previous section, the instances $equals_{[\mathsf{Int}]}$ and $equals_{\mathsf{Tree}}$ of equality, and properties such as for example

$$\begin{array}{ll}
\forall x & .\ equals\{|\mathsf{t}|\}\ x\ x \\
\forall x\ y & .\ equals\{|\mathsf{t}|\}\ x\ y \Longrightarrow equals\{|\mathsf{t}|\}\ y\ x \\
\forall x\ y\ z & .\ equals\{|\mathsf{t}|\}\ x\ y \wedge equals\{|\mathsf{t}|\}\ y\ z \Longrightarrow equals\{|\mathsf{t}|\}\ x\ z.
\end{array}$$

Our tool generates a set of generic functions, all of which have the specified type or a more general type. The instances of the generic functions thus generated must agree with the example instances provided, and these must also satisfy the properties expected of the generic function.

A generic function consists of a number of cases that define the behaviour of the function for a specific type. Most generic functions in the Generic Haskell library have cases for base types such as Int, Float, Char, and Double, the sum type $+$, the product type $\times$, Unit, the types Con, and Label for representing constructors and record labels, respectively, and sometimes also $\rightarrow$, IO, etc. Not all of these cases are required: in principle, providing cases for $+$, $\times$, and Unit is sufficient if no base types are used in the data types on which a generic function is used. When Con and Label cases are not provided, the Generic Haskell compiler generates default definitions.

For the purposes of this chapter, we assume that the generated generic functions have cases for the types Bool (as a very simple representative of base types), $+$, $\times$, and Unit.

Since types play a prominent rôle in the definition of generic functions, types drive the generation of our functions. Our tool instantiates the type of the generic function to each base type for which cases are normally defined. Then the instantiated types are used with Djinn to generate the corresponding cases. The cases thus obtained, are then

tested to check whether these satisfy the properties expected of the generic function. At this point, we have sets of case definitions for each base type. Our tool obtains a set of generic functions by applying the cartesian product on these sets.

Finally, the definitions of generic functions thus obtained are pruned by testing them against the instance examples provided by the user. The generic functions are specialized on the types of instance examples and each specialization is tested for equality against the examples by means of random testing.

6.4.1 Djinn

In this chapter we use Djinn for the type-based generation of terms. This tool implements a decision procedure for intuitionistic propositional calculus due to Dyckhoff (1992), and, effectively exploiting the Curry-Howard correspondence, uses it to solve the type-inhabitation problem for the simply-typed lambda calculus.

To an end user, Djinn looks like the interactive environment of the Glasgow Haskell Compiler. A user can request the generation of a term by writing an identifier followed by a question mark and the type to be inhabited:

```
Djinn> id ? a -> a
id :: a -> a
id x1 = x1
Djinn> const ? a -> b -> a
const :: a -> b -> a
const x1 _ = x1
```

More complex queries can involve functional arguments to define, for example, the composition of functions, currying and uncurrying:

```
Djinn> o ? (b -> c) -> (a -> b) -> a -> c
o :: (b -> c) -> (a -> b) -> a -> c
o x1 x2 x3 = x1 (x2 x3)
Djinn> curry ? ((a, b) -> c) -> a -> b -> c
curry :: ((a, b) -> c) -> a -> b -> c
curry x1 c3 c4 = x1 (c3, c4)
Djinn> uncurry ? (a -> b -> c) -> (a, b) -> c
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry x1 (v3, v4) = x1 v3 v4.
```

At any point the user can introduce new symbols in the environment by giving a data type declaration.

```
Djinn> data Maybe a = Nothing | Just a
Djinn> maybe ? b -> (a -> b) -> Maybe a -> b
maybe :: b -> (a -> b) -> Maybe a -> b
maybe x1 x2 x3 =
        case x3 of
```

```
Nothing -> x1
Just v6 -> x2 v6
```

So far we have seen only unique inhabitants of a specific given type. But this need not be the case:

```
Djinn> choose ? a -> a -> a
choose :: a -> a -> a
choose _ x2 = x2
-- or
choose x1 _ = x1.
```

Djinn can generate a number of terms for a type, but this does not mean that all possible terms are generated. We discuss limitations of Djinn in a later subsection.

6.4.2 Type-based term generation

Given an environment Γ and a type t, a type-based term-generation tool such as Djinn generates a set of terms E, such that every term e in E has type t: $\Gamma \vdash e :: \mathsf{t}$. The size of this set is potentially infinite, thus tools that generate such definitions restrict either the maximal size of such a set or the maximal size of generated terms.

Our approach makes use of a type-based generation tool to generate case definitions for a generic function. The tool is applied to the different base-type instantiations of the function signature. For example, this is the signature instantiation of *equals* for sum types:

$$\begin{aligned} \mathit{equals}\{\!|\alpha+\beta|\!\} :: \forall \mathsf{a}\ \mathsf{b}\,.\, & (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to \\ & (\mathsf{b} \to \mathsf{b} \to \mathsf{Bool}) \to \\ & (\mathsf{a}+\mathsf{b}) \to (\mathsf{a}+\mathsf{b}) \to \mathsf{Bool}. \end{aligned}$$

Due to the self-dependency (the recursive call of *equals* to itself), the signature describes a higher-order function that takes equality functions to compare the sum components.

There are infinitely many terms of this type. We show a few of them:

$$
\begin{array}{lll}
\lambda f\ g\ x\ y & \to \mathit{True} \\
\lambda f\ g\ x\ y & \to \mathit{False} \\
\lambda f\ g\ s_1\ s_2 & \to \textbf{case}\ s_1\ \textbf{of} \\
& \quad \mathit{Inl}\ x & \to \textbf{case}\ s_2\ \textbf{of} \\
& & \quad \mathit{Inl}\ y \to \mathit{False} \\
& & \quad \mathit{Inr}\ y \to g\ y\ y \\
& \quad \mathit{Inr}\ x & \to \textbf{case}\ s_2\ \textbf{of} \\
& & \quad \mathit{Inl}\ y \to f\ y\ y \\
& & \quad \mathit{Inr}\ y \to \mathit{True}. \\
\lambda f\ g\ s_1\ s_2 & \to \textbf{case}\ s_1\ \textbf{of} \\
& \quad \mathit{Inl}\ x & \to \textbf{case}\ s_2\ \textbf{of} \\
& & \quad \mathit{Inl}\ y \to f\ x\ y \\
& & \quad \mathit{Inr}\ y \to \mathit{False} \\
& \quad \mathit{Inr}\ x & \to \textbf{case}\ s_2\ \textbf{of} \\
& & \quad \mathit{Inl}\ y \to \mathit{False} \\
& & \quad \mathit{Inr}\ y \to g\ x\ y
\end{array}
$$

The three first terms have the same type as equality on sums but they do not have the right behaviour. The last term implements equality on sums. The type of equality on sums is not small enough to be inhabited only by equality. For example, Bool has constant values *True* and *False*, so it is possible to generate trivial terms that do not define equality. The third term contains applications such as $f\ y\ y$, which are undesirable for equality. It is possible to discard the third definition and hence reduce the search space by slightly generalizing the signature of the generic function. To illustrate the idea, consider the signature of the generic map in Generic Haskell:

$$\mathit{gmap}\{\!|\mathsf{a},\mathsf{b} :: *|\!\} :: (\mathit{gmap}\{\!|\mathsf{a},\mathsf{b}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{b}.$$

The type of generic map on sum types is as follows:

$$\mathit{gmap}\{\!|\alpha+\beta|\!\} :: \forall \mathsf{a}\ \mathsf{b}\ \mathsf{c}\ \mathsf{d}\,.\,(\mathsf{a}\to\mathsf{c}) \to (\mathsf{b}\to\mathsf{d}) \to \mathsf{a}+\mathsf{b} \to \mathsf{c}+\mathsf{d}.$$

Due to the general type of $\mathit{gmap}\{\!|\cdot|\!\}$, there is very little choice in the terms that can be generated. The target sum value can only be produced by using the functional arguments, and these can only be applied to the input sum components. Therefore we get the following term:

$$
\begin{array}{lll}
\mathit{gmap}_+\ f\ g\ s = \textbf{case}\ s\ \textbf{of} \\
\quad \mathit{Inl}\ x & \to \mathit{Inl}\ (f\ x) \\
\quad \mathit{Inr}\ x & \to \mathit{Inr}\ (g\ x).
\end{array}
$$

There is still an infinite number of terms having this type, but (assuming total functions) the other terms are equivalent. Examples of such terms are repeated applications of identity or other expressions having no effect on the result of the function. Interestingly, Djinn does not generate such terms containing "useless" expressions. The idea of reducing the search space by generalizing the signature of generic functions is elaborated further in Section 6.4.4.

6.4.3 Limitations of Djinn

Recursive data types

In Section 6.4.1 we have shown that Djinn can also generate functions for types involving user-defined data types such as Maybe a. However, Djinn does not support recursive data types such as lists or trees. While this restricts the family of functions that can be generated, this is not a big problem for our approach, since none of the base types is recursive. Moreover, the generated generic functions can be specialized on a recursive data type using the specialization algorithm of Generic Haskell. However, if our set of base types would include one or more recursive data types, it would be a problem. For example, a Djinn-based approach cannot be used to generate generic functions using the fixed-point view (Holdermans et al. 2006), because it includes the recursive base type Fix. Furthermore, this Djinn restriction also rules out generating generic functions with recursive types in their signatures. For example, since function *enum* returns a *list* of values:

$$enum\{\!|\mathsf{a} :: * |\!\}\ ::\ (enum\{\!|\mathsf{a}|\!\}) \Rightarrow [\mathsf{a}],$$

we cannot use Djinn to generate it.

Incompleteness

In order to ensure termination for every query, Djinn implements a sequent calculus that does not have a contraction rule. This design allows Djinn to answer the type-inhabitation problem in finite time. The price to be paid is that not all terms inhabiting a type are generated. For instance, the type below corresponds to Church numerals.

```
Djinn> num ? (a -> a) -> a -> a
num :: (a -> a) -> a -> a
num x1 = x1
-- or
num x1 x2 = x1 x2
-- or
num _ x2 = x2
```

While the set of Church numerals is infinite, Djinn only generates the terms corresponding to zero and one. For this reason, we have refrained from trying to generate the Int cases of generic functions.

6.4.4 Generalizing the type of a generic function

The set of terms generated by Djinn for an argument type may be very large. The size of this set is the limiting factor in our approach to generating generic functions. We apply two steps in our approach to restrain the size of the set of generated terms. The first step we apply is using the observation made earlier: the more general the type of a function, the fewer terms are generated. It is easy to see why this is the case for the

following example. Given an a it is easy to construct an a by means of the identity function, so asked for constructing a term of type $\mathsf{a} \to \mathsf{a}$, Djinn returns one term: the identity function. It is impossible to construct a value of type b out of a value a, so the set of values of type $\mathsf{a} \to \mathsf{b}$ returned by Djinn is empty. Generalizing a type where occurrences of the same variable are replaced by different variables leads to fewer terms generated by Djinn, since there are fewer input terms to choose from for the output.

Before we let Djinn generate terms for the types obtained by instantiating the generic type on the base types, we first generalize the type of the generic function. For example, while the signature we gave in Section 6.2 for equality is

$$\mathit{equals}\{\!|\mathsf{a} :: *|\!\} :: (\mathit{equals}\{\!|\mathsf{a}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool},$$

we replace it now by the more general

$$\mathit{equals}\{\!|\mathsf{a},\mathsf{b} :: *|\!\} :: (\mathit{equals}\{\!|\mathsf{a},\mathsf{b}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{b} \to \mathsf{Bool},$$

that is, we replace occurrences of the same variable in the type by different variables. Now, the type for the + case of equality is

$$\begin{aligned}\mathit{equals}\{\!|\alpha+\beta|\!\} :: \forall \mathsf{a}\ \mathsf{b}\ \mathsf{c}\ \mathsf{d}\,.\, &(\mathsf{a} \to \mathsf{c} \to \mathsf{Bool}) \to \\ &(\mathsf{b} \to \mathsf{d} \to \mathsf{Bool}) \to \\ &(\mathsf{a}+\mathsf{b}) \to (\mathsf{c}+\mathsf{d}) \to \mathsf{Bool}.\end{aligned}$$

Using this type, Djinn generates a non-empty, and much smaller set of terms than for the original type of $\mathit{equals}\{\!|\alpha+\beta|\!\}$.

Generalizing the type of a generic function has the added advantage that the function becomes more flexible. For example, the generalized equality can also compare list of characters with list of integers (for example representing entries in the ASCII table) by supplying the function *ord*. This type of generalization is very common in generic programming, for example, the type of `pequal` in PolyLib (Jansson and Jeuring 1998b), the library of PolyP, is $(\mathsf{a} \to \mathsf{b} \to \mathsf{Bool}) \to \mathsf{d}\ \mathsf{a} \to \mathsf{d}\ \mathsf{b} \to \mathsf{Bool}$.

The type of a generic function can always safely be generalized: the general terms thus generated can also be type-checked with the former, more specific types.

We leave the generalization step to the user of our tool. However, automating this step is very simple, so we might add the option to a future version of our tool.

6.4.5 Pruning generated terms by property

The second step in decreasing the number of terms generated by Djinn is to prune the set of terms obtained from Djinn by means of the properties specified for a generic function.

For example, we want the equality function to be reflexive, symmetric, and transitive. Since it is in general impossible to automatically *prove* properties of functions, we *test* these properties instead. So for each of the four type indices, the functions generated by Djinn are tested whether or not they satisfy the instance of the properties of equality on the type. For the symmetry property on the Bool type we test the following property:

$$\forall x\, y\,.\, equals\{|\mathsf{Bool}|\}\ x\ y == equals\{|\mathsf{Bool}|\}\ y\ x.$$

This is a simple instance of the general symmetry property for equality. For the sum type, which has kind $* \to * \to *$, the instance of the property is a bit more involved:

$$\begin{array}{l}\forall eqa\ x\ y\,.\, eqa\ x\ y == eqa\ y\ x \Longrightarrow \\ \quad \forall eqb\ v\ w\,.\, eqb\ v\ w == eqb\ w\ v \Longrightarrow \\ \quad\quad \forall s\ t\,.\, equals\{|\mathsf{a}+\mathsf{b}|\}\ eqa\ eqb\ s\ t == equals\{|\mathsf{a}+\mathsf{b}|\}\ eqa\ eqb\ t\ s,\end{array}$$

where the dependency of $equals\{|\mathsf{a}+\mathsf{b}|\}$ on *equals* has been made explicit.

As it happens, the four equality functions on the sum type generated by Djinn given in Section 6.4.2 satisfy the above property. A generated term that does not satisfy it is:

$$\begin{array}{l}\lambda x\ y\ z \to \mathbf{case}\ z\ \mathbf{of} \\ \qquad Inl\ v \ \to \lambda s \to \mathbf{case}\ s\ \mathbf{of} \\ \qquad\qquad\qquad\qquad Inl\ m \to x\ v\ m \\ \qquad\qquad\qquad\qquad Inr\ \_ \to False \\ \qquad Inr\ w \to \lambda t \to \mathbf{case}\ t\ \mathbf{of} \\ \qquad\qquad\qquad\qquad Inl\ \_ \to True \\ \qquad\qquad\qquad\qquad Inr\ n \to y\ w\ n.\end{array}$$

Just as with types of type-indexed functions, properties on type-indexed functions depend on the kind of the type arguments (Hinze 2000a). It suffices to specify the property on types of kind $*$, such as the property for $equals\{|\mathsf{Bool}|\}$ above; the properties for types with higher-order kinds can then automatically be generated.

At the moment the instances of properties on particular types have to be written by hand; we expect that in the near future we will add kind-indexed property generation to our tool.

6.4.6 QuickCheck

We use the QuickCheck library (Claessen and Hughes 2000) for testing whether or not properties hold, and for testing equality of example instances provided by the user and specialized instances obtained from the generated generic functions. QuickCheck tries to falsify properties specified by the user. It typically generates 100 random test cases, which are used to test a property. One of the properties we define is the following:

$$\begin{array}{l}prop\_EqFuns\ specializedFun\ exampleFun \\ \quad = \lambda x \to specializedFun\ x == exampleFun\ x.\end{array}$$

This property is parametrized with the specialized generic function and the example function specified by the user. It tests the equality of the two functions for an arbitrary argument. Note that this property applies to functions of one argument; functions with more arguments would take additional x's. A property like this, properly instantiated with the example and the generic function, is passed to the *run* function in QuickCheck,

and QuickCheck then randomly generates several values of x to test the validity of the property.

Not all properties are easy to check using QuickCheck. Since QuickCheck randomly generates values, it is rather unlikely that it generates equal values if we test on a data type that contains many values. So the transitivity property of equality, which takes three random values x, y, and, z, and checks that x and z are equal whenever x and y are equal and y and z are equal, is unlikely to ever check that x and z are equal. Special measures have to be taken in such cases, such as for example restricting the type of values to small types, or to write special generators.

6.5 Implementation

This section discusses an implementation of our tool, and the design choices we made to implement it.

As outlined in Section 6.4, our approach consists of a generation phase, based on Djinn, and a couple of pruning phases. The terms generated for the type indices of generic functions are pruned by checking their validity against properties. The set of generic functions obtained by taking the cross product of the type-indexed terms, is pruned by checking the equivalence of specializations with respect to user-provided instances. Since, in general, there is no algorithm that can prove the validity of a term with respect to a property or an instance, we turn to type-based automatic testing to test properties and the equivalence with user-provided examples.

Besides the functionality offered by Djinn, our approach requires specialization and testing. A very straightforward approach would be to pretty print the terms generated by Djinn to Generic Haskell source files, compile these files using the Generic Haskell compiler, and then run the Glasgow Haskell Compiler to produce a binary that tests and reports which properties pass or fail for which generated functions. The advantage of this approach is that we can use the specialization algorithm from Generic Haskell for specializing generic functions and signatures, and the automated testing library QuickCheck. This approach, however, is likely to incur significant overhead due to the generation and compilation of code for *every* candidate function. This problem becomes more serious if we take into account that we may generate and prune thousands of expressions for a given generic-function signature.

An alternative approach is to write an expression interpreter integrated with Djinn. We then would have to write specialization functionality and QuickCheck functionality for this expression language. Although the interpreter approach does not suffer from compiler overheads, it is a considerable task to implement (subsets of) the languages and libraries of Generic Haskell, Haskell, and QuickCheck.

In this chapter we use a more sophisticated variant of the interpreter approach. Instead of directly interpreting the terms generated by Djinn, we first transform terms into well-typed terms. In this way it becomes possible to implement an efficient interpreter that does away with the tagging and untagging that an untyped interpreter needs. Generalized algebraic data types (Xi et al. 2003; Cheney and Hinze 2003; Peyton Jones

et al. 2006) (GADTs) play an essential rôle here. Moreover, since the interpreted values are Haskell values, it is possible to use the QuickCheck library. In the rest of this section we describe our typed interpreter and how it integrates with automated testing using QuickCheck and with the specialization of generic functions.

6.5.1 Expressions and types

Although the generation algorithm of Djinn is type based, which guarantees that generated terms are well-typed expressions, the type information is not preserved in the terms. So the terms we obtain from Djinn are untyped. To use these terms in Haskell, we have to evaluate them into untagged Haskell values. These Haskell values can then directly be tested against properties and instances using QuickCheck.

As usual, expressions consist of variables, lambda abstractions, applications and constants.

```
data Expr = EVar String
          | ELam String Expr
          | EApp Expr Expr
          | ECon String
```

As an example, here is how we write the *swap* function as a value of type Expr:

```
swap = ELam "x"
         (EApp (EApp (ECon "(,)")
                     (EApp (ECon "snd")
                           (EVar "x")))
               (ECon "fst" `EApp` EVar "x")).
```

Types are either variables, constants, or the application of a type to a type.

```
data Type = TVar String
          | TCon String
          | TApp Type Type
```

We do not extend the syntax of expressions to cover generic functions as well, but instead represent generic functions by labeled records that contain the cases for the different type indices Bool, Unit, $\times$, and $+$ for generic functions.

```
data GenDefinition =
     GenDefinition{ bool :: Expr
                  , unit  :: Expr
                  , prod  :: Expr
                  , sum   :: Expr
                  }
```

A signature of a generic function consists of a name, a list of generic type variables (before type generalization the equality function has one generic type variable, *gmap* has two), and a type.

```
data GenSignature =
     GenSignature{ genName :: String
                 , genVars  :: [String]
                 , genType  :: Type
                 }
```

We assume generic functions depend on themselves and not on other generic functions. The first assumption does not restrict the domain: very few generic functions are not self dependent (Löh 2004), and dependencies do not have to be used. The second assumption does restrict the applicability of the tool. However, allowing dependencies on other generic functions is not difficult.

A generic function can be specialized on a type. We have functions for specializing generic signatures and functions:

```
specGenSignature :: GenSignature → Type → Type
specGenDefinition :: GenSignature → GenDefinition → Type → Expr.
```

The definitions of these functions are based upon the standard specialization algorithms for Generic Haskell, which have been described in several places (Hinze 2000c; Hinze and Jeuring 2003; Löh 2004), and are therefore omitted from this chapter.

6.5.2 Typing untyped terms

To turn Djinn-generated terms into untagged Haskell values we first have to type them. This section briefly explains how we type the untyped terms generated by Djinn. Our approach is reminiscent of the dynamic typing solution given by Baars and Swierstra (Baars and Swierstra 2002) for a similar problem. The main difference is that we use De Bruijn indices for variables, making it impossible to fail when looking up a variable.

Typed terms are defined as a GADT to enforce the constraints that make terms well-typed.

```
data Zero
data Succ a
data Lookup :: ∗ → ∗ → ∗ → ∗ where
  ZL :: Lookup Zero (a, env) a
  SL :: Lookup i env a → Lookup (Succ i) (x, env) a
data TExpr :: ∗ → ∗ → ∗ where
  TApp    :: TExpr env (a → b) → TExpr env a → TExpr env b
  TVar    :: Lookup i env a → TExpr env a
  TLambda :: TExpr (a, env) b → TExpr env (a → b)
evaluate :: TExpr () a → a
```

This presentation is heavily inspired by the representation of typed terms in dependently typed programming languages (McBride 2003). The TExpr data type represents

a valid typing judgement that follows the structure of a typed term. The first argument is the environment under which the judgement holds and the second argument is the type assigned by the judgement to the term. Variables are represented using De Bruijn indices. The *TVar* constructor selects a type in the environment using the data type Lookup. If a type expression has an empty environment we can obtain the represented type by means of the function *evaluate*.

The function *typeInfer* recovers the type information for a term generated by Djinn.

```
typeInfer :: Expr → Maybe Typed
data Typed :: ∗ where
  Typed :: TExpr () a → Rep a → Typed
data Rep :: ∗ → ∗ where
  RU    :: Rep ()
  RBool :: Rep Bool
  RProd :: Rep a → Rep b → Rep (a, b)
  RSum  :: Rep a → Rep b → Rep (Either a b)
  RArr  :: Rep a → Rep b → Rep (a → b)
```

If function *typeInfer* manages to infer a type, it returns a value of type Typed. A Typed value contains an existentially quantified type and its representation. It is possible to recover this type by analyzing the type-representation witness. Since the type of the expression is existential, it is sometimes necessary to compare it with another type representation to obtain a type-equality proof:

```
unify :: Rep a → Rep b → Maybe (TEq a b)
data TEq :: ∗ → ∗ → ∗ where
  TEq :: TEq a a.
```

We omit the definitions of *evaluate*, *unify* and *typeInfer*. The interested reader can find the ideas behind these definitions in the literature (Baars and Swierstra 2002; Xi et al. 2003; Cheney and Hinze 2003).

6.5.3 Term generation

The generation function has the following, simple, type:

```
generate :: Type → [Expr].
```

Our implementation of *generate* uses Djinn, but we could have used any of the approaches described in (Katayama 2005; Koopman and Plasmeijer 2006).

Type-based generation yields a set of functions for each of the type indices that make up a generic function. Hence, to obtain the generic function we combine all generated terms in a cross product, using a list comprehension.

```
generateGenDefinition :: GenSignature → [GenDefinition]
generateGenDefinition sig =
  [GenDefinition     {bool = boolDef
                     ,unit  = unitDef
                     ,prod  = prodDef
                     ,sum   = sumDef
                     }
   | boolDef ← boolTerms
  , unitDef  ← unitTerms
  , prodDef  ← prodTerms
  , sumDef   ← sumTerms
  ]
  where
    boolTerms = genTypeCase tBool
    unitTerms = genTypeCase tUnit
    prodTerms = genTypeCase tProd
    sumTerms = genTypeCase tSum
    genTypeCase = generate ∘ specGenSignature sig,
```

where *tBool*, *tUnit*, *tProd*, and *tSum* are values of type Type representing the types Bool, Unit, $\times$, and $+$, respectively.

Since Djinn sometimes generates many terms, it is important to first prune the terms generated by *genTypeCase* by property. The implementation of pruning is discussed in the following sections.

Note that if the generator returns many terms for the type indices, huge structures are built here. The cross-product function is the bottleneck of our approach.

6.5.4 Testing properties of functions

To prune the set of candidate functions, we test generated terms against user-provided properties, and specializations of generated generic functions against user-provided example instances. We use QuickCheck for testing, so we have to construct QuickCheck properties for pruning.

```
data Prop = ∀a b . Testable b ⇒ Prop (Rep a) (a → b)
```

A property consists of a representation of the type a, and a function that takes a value of type a and returns a value which, when tested with QuickCheck, determines whether or not the original value of type a is valid. In general, a property takes the form of a function that applies the value to be checked to test cases generated by QuickCheck. The type of the property (b) is an instance of the type class Testable to enable the automatic generation of test cases. As an example, consider the reflexivity property for the equality function on booleans and products of booleans:

$$
\begin{aligned}
&\mathit{reflEqBool}, \mathit{reflEqProdBool} :: \mathsf{Prop}\\
&\mathit{reflEqBool} \quad\;\; = \mathit{reflEq}\ \mathit{RBool}\\
&\mathit{reflEqProdBool} = \mathit{reflEq}\ (\mathit{RProd}\ \mathit{RBool}\ \mathit{RBool})\\
&\mathit{reflEq} \quad :: (\mathsf{Testable}\ (\mathsf{a} \to \mathsf{Bool})) \Rightarrow \mathsf{Rep\ a} \to \mathsf{Prop}\\
&\mathit{reflEq}\ r = \mathit{Prop}\ (\mathit{sigEq}\ r)\ (\lambda f\ x \to f\ x\ x == \mathit{True})\\
&\mathit{sigEq} \quad :: \mathsf{Rep\ a} \to \mathsf{Rep}\ (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool})\\
&\mathit{sigEq}\ t = t\ \text{`}\mathit{RArr}\text{`}\ (t\ \text{`}\mathit{RArr}\text{`}\ \mathit{RBool}).
\end{aligned}
$$

The function that represents the reflexivity property, applied to equality on booleans, can be tested with QuickCheck because its type, $[\mathsf{Bool}] \to \mathsf{Bool}$, is an instance of type class Testable.

The testing function tests an untyped expression against a user-specified property:

$$\mathit{testProp} :: \mathsf{Prop} \to \mathsf{Expr} \to \mathsf{Bool}.$$

This function converts an expression to a Haskell value (Section 6.5.2) and, using QuickCheck, tests its validity with respect to the property. For example, the set of terms for booleans in the cross product is pruned as follows:

$$\mathit{boolTerms} = \mathit{filter}\ (\mathit{testProp}\ \mathit{reflEqBool})\ (\mathit{genTypeCase}\ \mathit{tBool}).$$

For type constructors the situation is a bit different. As explained in Section 6.4.5, a property for sums or products takes properties for its dependencies as argument. As a pragmatic solution we let the property depend on the first term for booleans that satisfies the property.

$$
\begin{aligned}
&\mathit{prodTerms} \quad = \mathit{filter}\ \mathit{pruneProd}\ (\mathit{genTypeCase}\ \mathit{tProd})\\
&\mathit{pruneProd}\ e = \mathit{testProp}\ \mathit{reflEqProdBool}\ (\mathit{appDep}\ e)\\
&\mathit{appDep}\ e \quad = e\ \text{`}\mathit{EApp}\text{`}\ \mathit{dep}\ \text{`}\mathit{EApp}\text{`}\ \mathit{dep}\\
&\mathit{dep} \qquad\;\; = \mathit{head}\ \mathit{boolTerms}
\end{aligned}
$$

Note that the cases of the generic function are pruned using properties *before* they are combined in the cross product. As a result, many unnecessary combinations that would otherwise be produced are no longer considered.

6.5.5 Pruning by example

Pruning by example compares the specialization of a candidate generic function on a data type for which we also have an example instance. For example, we might have equality on lists of integers as an example instance. If a counterexample, i.e. an argument for which the two functions return different results, is found, the candidate generic-function definition is discarded.

Each example contains a type index and a property that tests the function with an example:

```
data Example = Example Type Prop.
```

The type index requests a specialization of the generic function to be compared against the example function.

Here are some typical examples for the generic equality function:

```
[Example tBool (Prop (sigEq RBool) (λf x y → f x y == (x == y)))
,Example (tProd tBool tBool) (Prop (sigEq (RProd RBool RBool))
         (λf x y → f x y == (x == y)))
].
```

Given a generic signature, a generic function and an example we can determine the equivalence of the specialization of the generic function and the example by means of testing:

```
testGenDefinition :: GenSignature → [Example] → GenDefinition → Bool
testGenDefinition genSig examples genDef = and (map check examples)
  where check (Example tindex prop) =
          testProp prop (specGenDefinition genSig genDef tyindex).
```

The implementation of pruning for a list of generic functions involves filtering the list using *testGenDefinition* as predicate, just as for properties. Pruning by example, as the name suggests, prunes away the generic functions that are not extensionally equal to all the examples provided by the user.

```
pruneByExamples :: GenSignature → [Example] →
                   [GenDefinition] → [GenDefinition]
pruneByExamples genSig examples = filter (testGenDefinition genSig examples)
```

6.5.6 Termination of testing

In our approach we test terms generated by Djinn against properties, and we test specialized generic functions against provided examples. If such a test does not terminate, neither will our tool.

Testing terms generated by Djinn is not a problem, since Djinn only generates non-recursive functions, and termination is guaranteed.

Specializations generated for recursive types are recursive functions themselves, so here there is a possibility that a non-terminating specialization is generated, and that a test for such a function might not terminate.

Our tool does terminate for the subclass of generic functions consisting of so-called consuming (*equals*) and transforming (*gmap*, *zipWith*) generic functions. This subclass of generic functions is terminating because the recursive calls in the product and sum cases use the self-dependency functions, which can only take a component of the original product or sum as argument. So $equals\{\!|\alpha+\beta|\!\}$ can only be defined in terms of $equals\{\!|\alpha|\!\}$ and $equals\{\!|\beta|\!\}$. This ensures that at every recursive call the type argument

becomes smaller until a base case is reached (Unit or Bool) and the function terminates. Generating producing generic functions such as *empty* and *enum* (Löh 2004) may result in non-terminating generic functions. In future work we intend to generate this subclass of generic functions by imposing a timeout when performing tests.

6.6 Results

Experimentation with our tool has produced encouraging results with the generation of generic functions. We have generated generic equality, generic map, and generic zip. In this section we give an account of the attempts for each function.

To handle the generation of the generic equality function, we have generalized its signature to

$$equals\{|\mathsf{a},\mathsf{b}::*|\} :: (equals\{|\mathsf{a},\mathsf{b}|\}) \Rightarrow \mathsf{a} \to \mathsf{b} \to \mathsf{Bool},$$

as explained in Section 6.4.4. Furthermore, we supply the properties that specify that equality is reflexive, symmetric, and transitive, and we provide the instances of equality on lists of booleans and trees. Our tool generates the standard implementation of generic equality (a number of times in alpha-equivalent versions).

We have experimented a bit with taking subsets of the specification, all of which include the generalized type of equality.

First, only providing the examples is not always enough. For example, if we supply as example only equality on lists of booleans we also get a generic function with incorrect behavior for the sum case. The function returns *True* for two left components independent of whatever is in those components. This is because the left component of the sum in the structure type for lists is Unit, and the equality function for Unit always returns *True*. We had to wait for quite a while to obtain this definition: specifying properties not only helps in getting the right functions, we also get them (much!) faster.

Second, providing no examples, but only the properties, is not enough either. One of the functions generated for products only compares the first components of the products, and ignores the second components. However, the more properties are specified, the fewer solutions we get.

By far the easiest function to generate has been the generic map function. The highly polymorphic type of this function gives very little freedom to Djinn to generate terms. In fact, each of the following type queries in Djinn produces a unique correct answer:

$$\begin{array}{ll} gmap_{+} & ?\,(\mathsf{a} \to \mathsf{c}) \to (\mathsf{b} \to \mathsf{d}) \to \mathsf{a} + \mathsf{b} \to \mathsf{c} + \mathsf{d} \\ gmap_{\times} & ?\,(\mathsf{a} \to \mathsf{c}) \to (\mathsf{b} \to \mathsf{d}) \to \mathsf{a} \times \mathsf{b} \to \mathsf{c} \times \mathsf{d} \\ gmap_{\mathsf{Unit}} & ?\,\mathsf{Unit} \to \mathsf{Unit} \\ gmap_{\mathsf{Bool}} & ?\,\mathsf{Bool} \to \mathsf{Bool}. \end{array}$$

The only exception to this rule is the case for booleans, whose type leaves some freedom for term generation. The first solution returned by Djinn is the expected identity function. We enforce this choice by providing the property that map is the identity function on types of kind $*$ (such as Bool).

When trying to generate the generic *zipWith* function, a generalization of the standard *zipWith* function with signature

$$zipWith\{|\mathsf{a},\mathsf{b},\mathsf{c}::*|\} :: (zipWith\{|\mathsf{a},\mathsf{b},\mathsf{c}|\}) \Rightarrow \mathsf{a} \to \mathsf{b} \to \mathsf{c},$$

we encountered a problem with the sum case. Consider the sum case definition of *zipWith*, as given in Löh's thesis (Löh 2004)

$$\begin{aligned} zipWith\{|\alpha+\beta|\}\ (Inl\ x)\ (Inl\ y) &= Inl\ (zipWith\{|\alpha|\}\ x\ y) \\ zipWith\{|\alpha+\beta|\}\ (Inr\ x)\ (Inr\ y) &= Inr\ (zipWith\{|\beta|\}\ x\ y) \\ zipWith\{|\alpha+\beta|\}\ \_\ \ \ \_\ \ \ &= undefined. \end{aligned}$$

Using the analogy with lists, we can say that different alternatives correspond to lists of different lengths. This implementation, unlike the Haskell ones which returns an empty list, fails when structures of different shapes are zipped. To generate the sum case, Djinn needs to find a term of the following type:

$$(\mathsf{a} \to \mathsf{c} \to \mathsf{e}) \to (\mathsf{b} \to \mathsf{d} \to \mathsf{f}) \to \mathsf{a}+\mathsf{b} \to \mathsf{c}+\mathsf{d} \to \mathsf{e}+\mathsf{f}.$$

Djinn cannot populate this type because it cannot find a proof of the corresponding logical expression. This is not surprising, because there does not exist such a proof. Djinn only generates total functions and the *zipWith* case for sums cannot be total. We could get around this limitation by adopting Haskell's solution and using the *empty* function (Löh 2004) to generate a value corresponding to the empty list. However, we prefer to follow a simpler approach and change the type of *zipWith* so that it can cope with failure without depending on other functions.

$$zipWith\{|\mathsf{a},\mathsf{b},\mathsf{c}::*|\} :: (zipWith\{|\mathsf{a},\mathsf{b},\mathsf{c}|\}) \Rightarrow \mathsf{a} \to \mathsf{b} \to \mathsf{Maybe}\ \mathsf{c}$$

This is a variant of the type of *zipWith* used in PolyLib. With this type, and the examples of *zipWith* on lists and booleans, Djinn generates the correct definition of *zipWith*. In particular, it is not hard to find a term of type:

$$(\mathsf{a} \to \mathsf{c} \to \mathsf{Maybe}\ \mathsf{e}) \to (\mathsf{b} \to \mathsf{d} \to \mathsf{Maybe}\ \mathsf{f}) \to \mathsf{a}+\mathsf{b} \to \mathsf{c}+\mathsf{d} \to \mathsf{Maybe}\ (\mathsf{e}+\mathsf{f}).$$

Alternatively, the property

$$x == y \wedge zipWith\ x\ y == Just\ x \vee x \neq y \wedge zipWith\ x\ y == Nothing$$

suffices to find this definition of *zipWith*.

We think that *zipWith* in the library of Generic Haskell should be replaced by this version.

6.7 Conclusions

We have presented an approach to the automatic generation of generic functions. Our approach instantiates the signature of the generic function to each base type for which

cases are normally defined. Then the instantiated types are used with Djinn to generate the corresponding generic function cases. A safe and useful step a user almost always should take here is generalizing the type of the desired generic function: the more general a type, the fewer terms are generated by Djinn. We prune the set of terms returned by Djinn by testing them against user-specified properties using QuickCheck. We further reduce the set of generic functions by testing specializations of generated generic functions against user-provided examples. We have shown that type-based generation of a number of generic functions is realistic. Our approach has a number of limitations, which are listed in the future work section below.

6.7.1 Related work

Koopman and Plasmeijer (2006) use a type-based systematic enumeration of terms for the problem of function generation. Function terms are represented in a system of data types such that all the values that are generated are well typed and follow syntactic restrictions that prevent the generation of non-terminating functions. Candidate functions are tested against input-output pairs until a function satisfying those tests is found. Enforcing type correctness in the syntax of expressions gives a simple and efficient approach for function generation. However, it is not clear how to extend this approach to generate functions that take functions as arguments, such as the sum and product cases of generic functions. We believe that our typed term representation makes it possible to use the enumeration approach with higher-order functions.

Katayama (2005) provides another approach to the systematic generation of lambda expressions. The enumeration takes into account type information when doing a breadth-first search of expressions. Recursion is only possible by means of paramorphism operators on lists and natural numbers. Because this is a general approach, we believe that the performance improvement techniques can also be applied to our tool, should we choose to pursue the systematic enumeration direction.

6.7.2 Future work

The current tool is a proof of concept, with which we have shown that it is possible to automatically generate generic functions from their type and example instances. We simplified our domain in a number of places: generic functions are self dependent, and cannot have dependencies on other generic functions. These restrictions are easy to lift. The fact that Djinn generates a finite number of terms for a type implies that it will be hard to generate Int or String cases using Djinn. Furthermore, since Djinn cannot handle recursive types, we cannot handle generic functions with recursive types in their signature. Lifting these restrictions requires adapting Djinn in a fundamental way. We will investigate to what extent this is possible. Since QuickCheck can only check monomorphic properties, example instances of generic functions have to provided for types of kind $*$. The type-specialization and function-specialization algorithms in our tool have only been implemented for types of kind $*$.

We intend to investigate using the systematic enumeration of typed terms as in the work of Koopman and Plasmeijer (2006), instead of, or besides, Djinn.

Our approach uses type-based generation, and example-based testing. It would be interesting to see if example-based generation with type-based testing would work equally well. We have experimented with example-based generation, in which we try to generate cases in the definition of a generic function from example instances. Our *ad-hoc* attempts were not very successful. We want to investigate whether or not it is possible to "invert" the fusion algorithm from Alimarine and Smetsers (2004), which fuses embedding-projection pairs with type-indexed functions to obtain code that is almost equal to hand-written example instances of generic functions, to obtain cases in the definition of a generic function.

To get access to constructor names, structure types in Generic Haskell contain occurrences of the Con type:

$$\textbf{data}\ \mathsf{Con\ a} = \mathit{Con}\ \mathsf{a}$$

The compiler tags occurrences of the Con type with information about the constructor, such as its name and arity. In a generic function we get access to this information in the Con case, for example (from the library of Generic Haskell):

$$\mathit{constructorOf}\,\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (\mathit{Con}\ a) = c$$

Of course, Djinn cannot generate functions based on constructor information. So generic functions that depend on constructor information are out of reach for the approach described in this chapter. The example-based generation type-based testing approach might not suffer from this problem.

Acknowledgements
Andres Löh discussed this question with us years ago. Kasper Brink and Remco Burema contributed to our discussions when we started the research reported in this chapter. Remco Burema and Stefan Holdermans commented on previous versions of this chapter. Anonymous referees provided useful feedback. Thanks are due to Lennart Augustsson for implementing Djinn, on which our work is fundamentally based.

Samenvatting

Functionele programmeertalen bewijzen zich steeds meer als effectief middel om de productiviteit van programmeurs te verhogen. Programmeurs werken sneller omdat ze op een hoger abstractieniveau kunnen programmeren, en besteden minder tijd aan het debuggen van de code omdat een functionele taal kan garanderen dat typefouten niet optreden tijdens de programma executie. De twee belangrijkste kenmerken die dit mogelijk maken zijn hogere-orde functies en parametrisch polymorfe typesystemen.

De bouwstenen van abstractie worden gevormd door functies. Als we een veelvoorkomend stuk code identificeren kunnen we dit in een functie vangen en abstraheren over de verschillen door deze tot argumenten van de functie te maken. Aangezien functies zelf argumenten van andere functies kunnen zijn kunnen deze verschillen willekeurige berekeningen zijn. Dit soort functies zijn zogenoemde hogere-orde functies, en maken het mogelijk om op een hoog niveau te abstraheren over berekeningspatronen. Functionele programma's zijn veelal beknopt en betrouwbaar omdat de toepassing van hogere orde functies veel omslachtige en foutgevoelige programmafragmenten overbodig maakt. De functie *fold* is een bekend voorbeeld voor functionele programmeurs. Veel recursieve berekeningen over lijsten kunnen vervaardigd worden door *fold* te voorzien van de juiste argumenten. Omdat *fold* het aflopen van de lijst implementeert hoeft deze expliciete recursie niet uitgeprogrammeerd te worden bij het schrijven van de functie op de lijst. Op die manier is door het gebruik van *fold* de kans kleiner dat per ongeluk een niet terminerende functie geschreven wordt.

Simpel gezegd zorgt statische typecontrole ervoor dat een functie nooit wordt toegepast op een niet compatibel argument. Tijdens het uitvoeren van een correct getypeerd programma kan het bijvoorbeeld niet voorkomen dat een Booleaanse negatie wordt toegepast op een string. Typecontrole is in het bijzonder van belang voor hogere-orde functies. Zonder typecontrole zou het koppelen van een run-time typefout (bijvoorbeeld bij het uitvoeren van *fold*) aan de locatie in de source (een verkeerd getypeerd *fold* argument) neerkomen op detective werk. Tevens bieden moderne functionele typesystemen een vorm van type abstractie die we parametrisch polymorfie noemen. Een functie die van deze vorm van abstractie gebruik maakt noemen we een parametrisch polymorfe functie. Een parametrisch polymorfe functie abstraheert over het type van de waarden waarover een berekening uitgevoerd wordt. Het geabstraheerde type wordt een extra argument dat meegeleverd moet worden als de functie toegepast wordt. Bijvoorbeeld: *fold* voert een recursieve berekening uit over een lijst, maar heeft geen weet van het type van de elementen in de lijst. Daarom is het type van deze elementen een type-argument van *fold*. Om de functie *fold* te gebruiken passen deze het toe op het type van de lijstelementen en deze infomatie wordt door het typesysteem gebruikt om te controleren of de argumenten van *fold* dit elementtype aankunnen. Het gedrag van een parametrisch

polymorfe functie hangt echter niet af van het het type-argument. De functie *fold* loopt bijvoorbeeld altijd op dezelfde manier de lijst af, ongeachte het elementtype.

Datatype-generiek programmeren vergroot de kracht van functionele talen nog verder door de mogelijkheid om ook het gedrag van een functie af te laten hangen van zijn type-argumenten. Om deze functies te onderscheiden van parametrisch polymorfe functies gebruiken we de term generieke functies. Generieke functies maken het mogelijk om code te hergebruiken in een groter scala aan situaties dan voorheen mogelijk was. We kunnen nu bijvoorbeeld een *fold* functie schrijven, waarvan het type-argument bepaalt hoe de datastructuur afgelopen wordt: een *fold* op een lijst loopt de lijststructuur af, en een *fold* op een waarde van een type dat een boom beschrijft zal de boomstructuur aflopen.

Dit proefschrift gaat over het makkelijker maken van het toepassen van datatype-generiek programmeren in de praktijk, om zo de bruikbaarheid van functionele talen verder te vergroten. We behandelen verschillende beperkingen van de huidige generieke programmeertechnieken wat betreft uitdrukkingskracht en gebruiksgemak.

Motivatie en bijdragen

De gebruikelijke manier om een generieke functie te definiëren is door middel van inductie op de structuur (of vorm) van het type-argument. De eerste programmeertaal die zulke definities ondersteunde was PolyP (Jansson and Jeuring 1997). In PolyP zijn datatypes gemodelleerd door een eindig aantal type-gevallen die door generieke functies afgehandeld kunnen worden. Generieke functies bestaan zo uit een eindig aantal gevallen en kunnen gespecialiseerd worden voor een oneindig aantal datatypes. PolyP maakt het mogelijk een groot aantal bruikbare generieke programma's te definiëren, te weten recursie schema's zoals *fold* en varianten (Meijer et al. 1991), opwaartse en neerwaartse accumulaties (Bird et al. 1996; Gibbons 1998), unificatie (Jansson and Jeuring 1997), rewriting and matching functions (Jansson and Jeuring 2000), functies om subexpressies te selecteren (Steenbergen et al. 2008), pattern matching (Jeuring 1995), etc.

Een generieke functie is een waarde geïndexeerd door types. Het is echter ook mogelijk om een *type* te definiëren dat geïndexeerd is door types. Type-geïndexeerde types, geïntroduceerd door Hinze et al. (2004), worden gebruikt wanneer een generieke functie een datastructuur verwerkt die afhangt van het type waarover de functie generiek is. Een voorbeeld is de *zipper* datastructuur, die een puur functionele navigatie biedt over de waarde van een datatype. De ondersteunde operaties omvatten het afdalen in een kind, bewegen naar een kind van dezelfde ouder, of terugnavigeren naar de ouder. De zipper slaat de deelboom in focus op, samen met de context waarin deze deelboom voorkomt. Aangezien deze context informatie datatype specifiek is, hangt het type van de zipper datastructuur af van het datatype waarover genavigeerd wordt. Vanwege deze afhankelijkheid moet de generieke variant van de zipper gedefiniëerd worden als een type-geïndexeerd type. Type-geïndexeerde types kunnnen ook gebruikt worden voor andere toepassingen, zoals generiek herschrijven (Jansson and Jeuring

2000; Van Noort et al. 2008), generieke tries, en varianten van de zipper (Huet 1997; McBride 2001; Hinze et al. 2004; Morris et al. 2006; McBride 2008).

In PolyP zijn datatypes gemodelleerd als een toepassing van een dekpunt op een functor. Deze dekpunt benadering maakt de recursieve positie in een datatype expliciet bij het modelleren. Het is deze eigenschap die bovengenoemde toepassingen geïmplementeerd in PolyP en type-geïndexeerde types mogelijk maakt. Helaas is de dekpunt benadering beperkt tot reguliere datatypes, zoals lijsten en binaire bomen. Daardoor kan PolyP niet gebruikt worden voor een verzameling datatypes die naar elkaar verwijzen. Zulke systemen van (wederzijds recursieve) datatypes komen echter veel voor in de praktijk. Zo worden ze veelvuldig gebruikt in compilers om abstracte syntaxbomen te modelleren, waarbij elke datatype declaratie overeenkomt met een syntactische categorie.

In dit proefschrift laten we zien hoe de dekpunt benadering voor datatypes gegeneraliseerd kan worden zodat deze ook (wederzijds recursieve) systemen van datatypes aan kan. Deze techniek maakt het mogelijk om de toepassingen gebaseerd op PolyP en type-geïndexeerde types te gebruiken in een grotere reeks van situaties. We illustreren deze verbeterde dekpunt benadering door middel van een aantal voorbeelden: recursieschema's zoals folding en compos (Bringert and Ranta 2006), de zipper en generic matching.

Dit proefschrift behandelt ook een probleem dat optreedt bij de combinatie van type-geïndexeerde types en interfaces. In PolyP worden generieke functies gedefiniëerd in termen van de typestructuur. De argumenten die worden meegegeven aan deze functies zijn echter normale datatype waarden. Hieruit volgt dat kennis van de typestructuur alleen nodig is bij het definiëren van een generieke functie, maar niet bij het toepassen ervan. Dit maakt het mogelijk voor programmeurs die zelf geen kennis van generiek programmeren hebben om bibliotheken te gebruiken die intern generieke programmeertechnieken toepassen. Als een generieke functie echter een argument heeft met een type-geïndexeerd type, dan vergt de constructie van dat argument gedetailleerde kennis van de typestructuur en de type-geïndexeerde typedefinitie. Soms is het mogelijk om generieke hulpfuncties te definiëren die de programmeur afschermen voor deze details, maar er bestaan voorbeelden waarbij deze oplossing bijzonder omslachtig is. Generiek herschrijven is een toepassing waarbij dit probleem optreedt. Het type van de herschrijfregels is gedefiniëerd in termen van een type-geïndexeerd type. Hierdoor vergt de specificatie van een herschrijfregel een uitgebreid gebruik van hulpfuncties, danwel gedetailleerde kennis van de typestructuur en de type-geïndexeerde type definitie. Kort gezegd bestaat de kans dat gebruikers van een bibliotheek die gebruik maakt van type-geïndexeerde types kennis moeten nemen van de details van de implementatie van de bibliotheek.

Om dit probleem op te lossen, laten we zien hoe de waarden van een type-geïndexeerd type geconstrueerd kunnen worden zonder de gebruiker bloot te stellen aan de implementatiedetails wat betreft de structuur van de types. Dit maakt het mogelijk om een bibliotheek gebaseerd op type-geïndexeerde types te gebruiken zonder de implementatiedetails bloot te geven. Onze oplossing is gericht op de toepassing van generiek herschrijven. Hoewel de herschrijfregels geïmplementeerd zijn door middel van type

geïndexeerde types, laat onze techniek de gebruiker herschijfregels specificeren in termen van constructoren van het oorspronkelijke datatype. De gebruiker hoeft dus niet op de hoogte te zijn van de details van het uitbreiden van datatypes met metavariabelen. Hoewel onze oplossing specifiek is voor generiek herschrijven, zijn wij van mening dat zij ook bruikbaar is voor andere toepassingen van type-geïndexeerde types.

Andere bijdragen van dit proefschrift zijn: een karakterisering van Haskell bibliotheken voor generiek programmeren die uitdrukkingskracht en gebruiksgemak van de verschillende bibliotheken vergelijkt; de toepassing van generieke programmeertechnieken om termen van de simpel getypeerde lambda calculus te synthetiseren; en de automatische generatie van generieke functies uit door de gebruiker geleverde specificaties.

Acknowledgements

I am immensely indebted to my promotors Johan Jeuring and Doaitse Swierstra for giving me the opportunity to learn how to do research. Johan has many attributes that make him the ideal supervisor. He has always been patient and generous with his time. He taught me about research by working alongside me, and as a result I have greatly improved my confidence in my research and writing skills. I do not think I would have made it without his encouragement in the times of doubt. Thank you Johan!

Doaitse Swierstra has earned my sincere gratitude not only because of his detailed comments on my work, and his wild enthusiasm for our area of research, but also because he took care of me like family when I arrived to the Netherlands. He drove to Eindhoven to make sure that I would survive my new environment, and he generously gave me a bike so that I could start training for the ever impossible task of holding an umbrella while biking against strong winds and rain.

During these years I have been very lucky to collaborate with many very talented researchers: Eelco Dolstra, Alex Gerdes, Baastian Heeren, Stefan Holdermans, Patrik Jansson, Oleg Kiselyov, Andres Löh, Bruno Oliveira, Thomas van Noort, and Gideon Smeding. I wish to thank them for making me even more motivated about my work and for helping me improve my approaches to research in several ways. I also had many inspiring discussions with colleagues and members of the "reading club": Sean Leather, José Pedro Magalhães, Erik Hesselink, Chris Eidhof and Eelco Lempsing. I would also like to thank Pablo Azero who introduced me to the beauty of functional programming, and Martijn Schrage who kindly translated a summary of my research into Dutch.

I am very grateful to the members of the reading committee, Jan van Eijck, Ralf Hinze, Ralf Lämmel, Lambert Meertens, and Rinus Plasmeijer, who kindly took the time to carefully read my thesis. Lambert spotted several typos and inconsistencies that allowed me to make many improvements.

During my PhD period I did an internship that helped me grow both as a scientist and a programmer. I want to thank my internship mentors Simon Marlow and Simon Peyton-Jones.

The working environment at Utrecht University was open and friendly. I am very grateful to all my colleagues in the Software Technology group. The daily working routine was spiced up by frequent foosball competitions. I thank Americo, Arthur, Juan Francisco, Martijn, Reinier, Rodrigo and Sean for all the hours of loud fun.

I have had many great friends during these years. I thank Americo, Alessandro, Camilo, Dario, David, Despina, Gar Yein, Juan Francisco, Judith, Karina, Luca, Marcos, Martijn, Metka, Nora, Osmar, and Yuan Ju. My friends gave me many good laughs,

life lessons and their company was uplifting in tough times. I also thank Carola with whom, despite the distance, I keep a very close contact.

While living in the Netherlands, I was happy to be close to my mother and her husband Jean Michel. My family in Bolivia, although far away, always supported and believed in me unconditionally. I especially thank my late grandmother, my grandfather, my brother and my father. My brother inspired me many times because of his endless energy and optimism, and it was my father who initially introduced me to computer programming.

Bibliography

Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC'04*, volume 3125 of *LNCS*, pages 16–31. Springer-Verlag, 2004.

Artem Alimarine and Sjaak Smetsers. Improved fusion for optimizing generics. In Manuel V. Hermenegildo and Daniel Cabeza, editors, *Practical Aspects of Declarative Languages, 7th International Symposium, PADL 2005, Proceedings*, volume 3350 of *Lecture Notes in Computer Science*, pages 203–218. Springer-Verlag, 2005.

Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Generic Programming*, pages 1–20. Kluwer, 2003.

Frank Atanassow and Johan Jeuring. Inferring type isomorphisms generically. In *Mathematics of Program Construction, 7th International Conference, MPC 2004*, LNCS 3125, pages 32–53, 2004.

Lennart Augustsson. Announcing Djinn, version 2004-12-11, a coding wizard. Available from `http://permalink.gmane.org/gmane.comp.lang.haskell.general/12747`, 2005.

Arthur Baars and Doaitse Swierstra. Typing dynamic typing. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming, ICFP'02*, pages 157–166. ACM Press, 2002.

Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming: An introduction. In *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115, 1999.

Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

Jean-Philippe Bernardy, Patrik Jansson, Marcin Zalewski, Sibylle Schupp, and Andreas Priesnitz. A comparison of C++ concepts and Haskell type classes. In *ACM SIGPLAN Workshop on Generic Programming*, 2008.

Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Mathematics of Program Construction, 4th International Conference, MPC 1998*, volume 1422 of *LNCS*, pages 52–67. Springer-Verlag, 1998.

Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

C. Böhm and A. Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Christophe Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–95, 2001.

Björn Bringert and Aarne Ranta. A pattern for almost compositional functions. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP'06*, pages 216–226, 2006.

Neil C. C. Brown and Adam T. Sampson. Matching and modifying with generics. In Peter Achten, Pieter Koopman, and Marco T. Morazán, editors, *Draft Proceedings of the Ninth Symposium on Trends in Functional Programming (TFP), 2008*, pages 304–318, 2008. The draft proceedings of the symposium have been published as a technical report (ICIS-R08007) at Radboud University Nijmegen.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP'05*, pages 241–253. ACM Press, 2005.

Manuel M. T. Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Peyton Jones. Generic programming with type families. Available via `http://www.cse.unsw.edu.au/~chak/papers/tidt-slides.pdf`, 2008.

James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Haskell Workshop'02*, pages 90–104, 2002.

Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2000*, pages 286–279, 2000.

Dave Clarke and Andres Löh. Generic haskell, specifically. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 21–47. Kluwer, B.V., 2003.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1–3, 2007*, pages 315–326. ACM Press, 2007.

John Derrick and Simon Thompson. FORSE: Formally-Based Tool Support for Erlang Development. Project description, 2005. URL `http://www.cs.kent.ac.uk/projects/forse/`.

Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore, 1996.

Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807, 1992.

Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17(2):145–205, 2007.

Jeremy Gibbons. Design patterns as higher-order datatype-generic programs. In *ACM SIGPLAN Workshop on Generic Programming*, 2006.

Jeremy Gibbons. Polytypic downwards accumulations. In *Mathematics of Program Construction, 4th International Conference, MPC 1998*, volume 1422 of *LNCS*, pages 207–233. Springer-Verlag, 1998.

Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. The MIT Press, Cambridge, Massachusetts, 1997.

Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

Bastiaan Heeren, Johan Jeuring, Arthur van Leeuwen, and Alex Gerdes. Specifying strategies for exercises. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *Intelligent Computer Mathematics, MKM 2008, Birmingham, UK, July 28–August 1, 2008, Proceedings*, volume 5144 of *LNAI*, pages 430–445. Springer-Verlag, 2008.

Ralf Hinze. Generics for the masses. *Journal of Functional Programming*, 16:451–482, 2006.

Ralf Hinze. *Generic Programs and Proofs*. 2000a. Habilitationsschrift, Bonn University.

Ralf Hinze. Efficient generalized folds. In Johan Jeuring, editor, *Proceedings Workshop on Generic Programming, WGP 2000*, pages 1–16, 2000b. Utrecht Technical Report UU-CS-2000-19.

Ralf Hinze. Polytypic values possess polykinded types. In *Mathematics of Program Construction, 5th International Conference, MPC 1998*, volume 1837 of *LNCS*, pages 2–27. Springer-Verlag, 2000c.

Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 1–56, 2003.

Ralf Hinze and Andres Löh. "Scrap Your Boilerplate" revolutions. In *Proceedings of the 8th International Conference on Mathematics of Program Construction, MPC'06*, LNCS 4014, pages 180–208, 2006.

Ralf Hinze and Andres Löh. Generic programming, now! In Roland Backhouse et al., editors, *Datatype-Generic Programming*, LNCS, pages 150–208. 2007.

Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 51(2):117–151, 2004.

Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap Your Boilerplate" reloaded. In Philip Wadler and Masimi Hagiya, editors, *FLOPS'06*, LNCS 3945, 2006.

Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approches to generic programming in Haskell. In *Datatype-Generic Programming*, LNCS 4719, pages 72–149. 2007.

Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In *Mathematics of Program Construction, 8th International Conference, MPC 2006*, volume 4014 of *LNCS*, pages 209–234. Springer-Verlag, 2006.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *HOPL III*, pages 12–1–12–55, 2007.

Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

Graham Hutton and Jeremy Gibbons. The Generic Approximation Lemma. *Information Processing Letters*, 79(4):197–201, 2001.

Patrik Jansson and Johan Jeuring. PolyP — a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL'97*, pages 470–482, 1997.

Patrik Jansson and Johan Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In *Workshop on Generic Programming*, pages 33–45, 2000.

Patrik Jansson and Johan Jeuring. Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, 1998a.

Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

Patrik Jansson and Johan Jeuring. PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand, 1998b.

Johan Jeuring. Polytypic pattern matching. In *"Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA '95"*, pages 238–248, 1995.

Patricia Johann and Neil Ghani. Foundations for structured programming with GADTs. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'08*, pages 297–308, 2008.

Susumu Katayama. Systematic search for lambda expressions. In M. van Eekelen, editor, *6th Symposium on Trends in Functional Programming, TFP 2005*. Institute of Cybernetics, Tallinn, 2005. ISBN 9985-894-88-X.

Oleg Kiselyov. Smash your boilerplate without class and typeable. `http://article.gmane.org/gmane.comp.lang.haskell.general/14086`, 2006.

Oleg Kiselyov. Compositional gmap in syb1. `http://www.haskell.org/pipermail/generics/2008-July/000362.html`, 2008.

Oleg Kiselyov, Ralf Lammel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107, New York, NY, USA, 2004. ACM Press.

Pieter Koopman and Rinus Plasmeijer. Systematic synthesis of functions. In Henrik Nilsson, editor, *7th Symposium on Trends in Functional Programming, TFP 2006*, 2006.

Pieter Koopman and Rinus Plasmeijer. Systematic synthesis of λ-terms. In *Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, 2007.

Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. In R. Peña and T. Arts, editors, *IFL'02*, volume 2670 of *LNCS*, 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming, ICFP'04*, pages 244–255, 2004.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP'05*, pages 204–215, 2005.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 26–37, 2003.

Ralf Lämmel and Joost Visser. A Strafunski Application Letter. In *Proceedings of Practical Aspects of Declarative Programming, PADL 2003*, LNCS 2562, pages 357–375, 2003.

Ralf Lämmel and Joost Visser. Typed combinators for generic traversal. In Shiriram Krishnamurthi and C. R. Ramakrishnan, editors, *Proceedings Practical Aspects of Declarative Programming, PADL 2002*, volume 2257 of *LNCS 2257*, pages 137–154, 2002.

Ralf Lämmel, Joost Visser, and Jan Kort. Dealing with large bananas. In *Workshop on Generic Programming*, 2000.

Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In *Haskell'03*, pages 27–38, 2003.

Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In Olin Shivers, editor, *Proceedings of the International Conference on Functional Programming, ICFP'03*, pages 141–152. ACM Press, August 2003.

Andres Löh, Johan Jeuring, Thomas van Noort, Alexey Rodriguez, Dave Clarke, Ralf Hinze, and Jan de Wit. The Generic Haskell user's guide, Version 1.80 - Emerald release. Technical Report UU-CS-2008-011, Utrecht University, 2008.

Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'08*, pages 287–295, 2008.

Conor McBride. The derivative of a regular type is its type of one-hole contexts. `strictlypositive.org/diff.pdf`, 2001.

Conor McBride. First-order unification by structural recursion. *Journal of Functional Programming*, 13(6):1061–1075, 2003.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2007. doi: 10.1017/S0956796807006326.

Lambert Meertens. Calculate polytypically! In H. Kuchen and S. D. Swierstra, editors, *PLILP*, LNCS 1140, pages 1–16, 1996.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In *"Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA '91"*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, 1991.

Neil Mitchell and Colin Runciman. A static checker for safe pattern matching in Haskell. In *Trends in Functional Programming*, volume 6. Intellect, 2007a.

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Haskell'07*, 2007b.

Peter Morris. *Constructing Universes for Generic Programming*. PhD thesis, The University of Nottingham, 2007.

Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In *Types for Proofs and Programs*, LNCS. Springer-Verlag, 2006.

Matthew Naylor and Colin Runciman. Finding inputs that reach a target expression. In *SCAM'07*, pages 133–142, 2007.

Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In Ralf Hinze, editor, *ACM SIGPLAN Workshop on Generic Programming*, 2008.

Thomas van Noort. Generic views for generic types. Master's thesis, Utrecht University, 2008.

Ulf Norell and Patrik Jansson. Polytypic programming in Haskell. In *Implementation of Functional Languages*, volume 3145 of *LNCS*, pages 168–184. Springer-Verlag, 2004.

Bruno C. d. S. Oliveira and Jeremy Gibbons. TypeCase: A design pattern for type-indexed functions. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell, Tallinn, Estonia, September 30, 2005*, pages 98–109. ACM Press, 2005.

Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. In Henrik Nilsson, editor, *Trends in Functional Programming*, pages 199–216, 2006.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP'06*, pages 50–61, 2006.

Simon Peyton Jones et al. The Haskell Prime language report – working draft, 2007. `http://hackage.haskell.org/trac/haskell-prime`.

Claus Reinke. Traversable functor data, or: X marks the spot. `http://www.haskell.org/pipermail/generics/2008-June/000343.html`, 2008.

Tim Sheard. Generic unification via two-level types and parameterized modules. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Florence, Italy, September 3–5, 2001*, pages 86–97. ACM Press, 2001.

Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *"Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA '93"*, pages 233–242, 1993.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Pittsburgh, Pennsylvania, 2002*, pages 1–16. ACM Press, 2002.

Martijn van Steenbergen, Jeroen Leeuwestein, Johan Jeuring, José Pedro Magalhães, and Sylvia Stuurman. Selecting (sub)expressions – generic programming without generic programs. Unpublished, 2008.

Doaitse Swierstra, Pablo Azero Alcocer, and João Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, 1999.

S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag, 1996.

Akihiko Takano and Erik Meijer. Longcut deforestation in calculational form. In *"Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA '95"*, pages 306–313, 1995.

Philip Wadler. Theorems for free! In *FPCA'89*, pages 347–359. 1989.

Stephanie Weirich. Higher-order intensional type analysis. In *In Proc. 11th ESOP, LNCS 2305*, pages 98–114. Springer-Verlag, 2002.

Stephanie Weirich. RepLib: a library for derivable type classes. In *Haskell'06*, pages 1–12, 2006.

Noel Winstanley and John Meacham. *DrIFT user guide*, 2006. `http://repetae.net/~john/computer/haskell/DrIFT/`.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL'03*, pages 224–235, New Orleans, January 2003.

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty

of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering,

Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for*

Clean. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

Curriculum Vitae

Alexey Luis Rodriguez Yakushev was born September 12 1979 in St. Petersburg, Russia. From 1998 to 2002, he studied computer science at Universidad Mayor de San Simon, Bolivia. From March 2002 to August 2002 he was an intern at Philips Research Laboratories at Eindhoven under the supervision of Lex Augusteijn. In September 2002, he started a Master in Software Technology at Utrecht University, which he finished in 2004. From 2004 to 2008 he was a PhD student at Utrecht University under the supervision of Johan Jeuring and Doaitse Swierstra. He also was an intern at Microsoft Research Cambridge from October to December 2006, under the supervision of Simon Marlow and Simon Peyton-Jones. Currently he works at Vector Fabrics B.V.