Exploring Generic Haskell

Generic Haskell van alle kanten
(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. W. H. Gispen, ingevolge het besluit van het College voor Promoties in het openbaar te verdedigen op donderdag 2 september 2004 des ochtends te 10.30 uur
door

Andres Löh

geboren op 18 augustus 1976 te Lübeck, Duitsland

promotoren: Prof. dr. Johan Th. Jeuring, Universiteit Utrecht
en Open Universiteit Nederland
Prof. dr. S. Doaitse Swierstra, Universiteit Utrecht

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics), and has been financed by the NWO (Nederlandse Organisatie voor Wetenschappelijk Onderzoek).

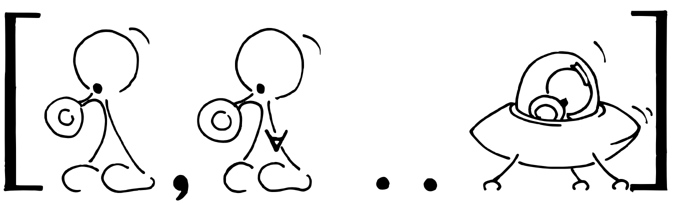
Printed by Febodruk, Enschede.

All drawings by Clara Strohm.

ISBN 90-393-3765-9

Contents

List of Figures



Acknowledgements

The story of me ending up in Utrecht to do a Ph.D. on Generic Haskell is full of lucky coincidences. Therefore, first of all, I want to thank fate, if there is such a thing, for the fact that everything turned out so well.

When I arrived in Utrecht, I did not even know my supervisor, Johan Jeuring – well, okay, I had read papers authored by him. Now, four years later, I can confidently say that I could not imagine a better supervisor. He has proved to be a patient listener, encouraged me where needed, warned me when I was about to be carried away by some spontaneous idea, let me participate in his insights, and shared his experiences regarding survival in the academic world. Most of all, he has become a friend. My whole time here would have been far less enjoyable if it were not for him. Johan, thank you very much.

Although on second position in these acknowledgments, Doaitse Swierstra may well have deserved the first. After all, he is "responsible" for the fact that I abandoned everything else, moved to Utrecht, and started working on a Ph.D. When I first met him at a conference, I asked him about possibilities to do a Ph.D. in Utrecht and to work in the area of functional programming. Afterwards, I had a hard time convincing him that it would not be possible for me to start immediately, but that I still had almost a year to go until finishing my "Diplom". During my time here, Doaitse has been a constant source of interesting ideas and enlightening discussions. I envy his seemingly unshakable enthusiasm for his work, and hope that this enthusiasm will inspire many others as it did inspire me.

I want to thank Ralf Hinze, who had the doubtful privilege to share an office with me during my first year, and as another German eased the transition to the Netherlands. For me, working on generic programming, he has been the ideal person to have close by and to learn from. I thank him for showing an interest in me and my work, also after he left.

Jeremy Gibbons, Ralf Hinze, Lambert Meertens, Rinus Plasmeijer, and Peter Thiemann are the members of the reading committee. I am grateful that they took the time to read this thesis, and for several helpful comments and insights. I am sorry that I could not act on all of them due to time constraints.

One of the really amazing things about Utrecht University has been the atmosphere in the software technology group. There are so many people that are interested in each other's work, and if there ever was a problem just outside my own area of expertise, I was very likely to find a helpful answer just a few steps away next door. Many of my colleagues have provided interesting ideas and valuable discussions. I would like to mention Daan Leijen, Eelco Dolstra, Arjan van IJzendoorn, and Bastiaan Heeren in particular, who have become good friends over the years.

I am very grateful to Martin Bravenboer, Bastiaan Heeren, Arjan van IJzendoorn, Daan Leijen, Ganesh Sittampalam, Ian Lynagh, Shae Matijs Erisson, André Pang, the "Matthiases" Auer and Weisgerber, Carmen Schmitt, Clara Strohm, and Günter Löh for reading parts of my thesis and constructively commenting on various aspects of my work. Especially to those of you who do not have a degree in computer science: thank you for taking the time and trying to understand what I have done.

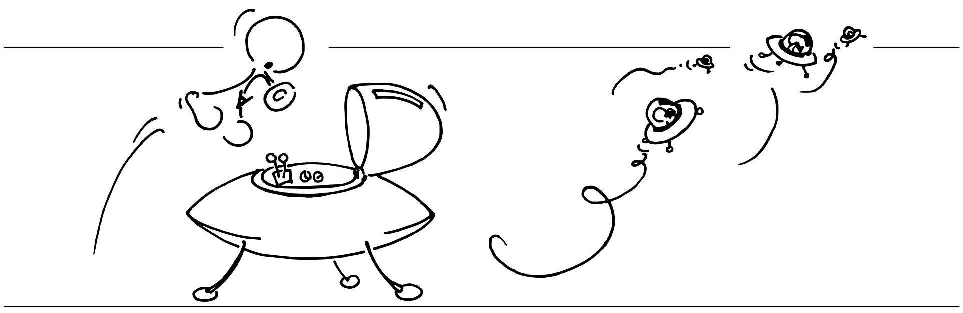
The people on the freenode `#haskell` channel, including, but not limited to earthy, Heffalump, Igloo, Marvin--, ozone, shapr, and SyntaxNinja, have provided a great deal of – probably largely subconscious – mental support during the phase in which I had to write my thesis and diversions were scarce.

I am indebted to Torsten Grust and Ulrik Brandes, who back during my time in Konstanz, managed to change my view of computer science into a positive one. Without them, I am convinced that I would never even have considered doing a Ph.D. in computer science. Thanks, Torsten, for exposing me to functional programming. I, too, once came to the university thinking that C is the only cool programming language in existence.

My parents deserve a big "Dankeschön" because they never complained about the choices I have made in my life, including the places where I have lived, and supported me in all my decisions in every way imaginable.

Words cannot describe what Clara means to me and how much impact she had on the creation of the thesis. Not only did she read almost all of it, finding several smaller and larger mistakes, she also invested an incredible amount of time and creativity into creating all the drawings in this thesis, thereby giving it a unique character – not to mention all the mental support and patience during a time where she herself had a thesis to write. To say it in the words the "Lambda" would choose: "Schlububius? Einmal? Alle Achtung!"

1 Adventure Calls!



This thesis is an exploration – an exploration of a language extension of the functional programming language Haskell. The extension is called *Generic Haskell*, albeit the name has been used to refer to different objects over the last several years: Many papers have described different proposals, features, variations, and generations of the language. One purpose of this thesis is to do away with at least part of this fuzziness: everything is described in a common notation and from a single starting point. The other purpose is to simply give a complete overview of the language: we will systematically explain the core features of Generic Haskell, and several extensions, all with motivating examples and details on how the features can be implemented.

Before we start our exploration, though, Section 1.1 will explain the idea and motivation behind *generic programming* which, at the same time, is the motivation for the design of Generic Haskell. After that, Section 1.2 will give an overview of the history of Generic Haskell. In Section 1.3 we discuss other important approaches to generic programming. In the last section of this chapter, Section 1.4, we give an overview of all the chapters of this thesis, their contents, the papers they are based on, and how they are interconnected.

1.1 From static types to generic programming

Static types are used in many programming languages to facilitate the creation of error-free software. While static types cannot guarantee the correctness of all programs written in a language, a sound static type system is capable of eliminating a certain class of runtime errors, which result in particularly nasty program crashes, because operating systems usually do not allow us to catch such errors. These errors result from the program inadvertently accessing a memory position that does not belong to the program. With static types, a program is checked at compile time, to prevent this kind of behaviour. Ideally, the successful type checking process together with the translation semantics of the language make up a proof that the programs cannot "go wrong" (Milner 1978).

In a less perfect world, these proofs often do not fully exist, because the translation that the compilers perform is too complex and involves too many external factors. Nevertheless, statically checked types in the user's programs result in less error-prone programs.

The amount of information about the program that a type checker can verify is not fixed. Some static type systems can type more programs than others, some can catch more errors than others – other kinds of errors, which are less nasty, but still inconvenient enough, such as divisions by zero or array indices that are out of bound. Nevertheless, there is a constant struggle: if too much cleverness is incorporated into the types, the type checking becomes inefficient or even undecidable. If too little information is covered by the type checker, programmers find themselves fighting the type system: programs that would behave correctly are nevertheless rejected as incorrect, because the type system is not capable of finding out that a potentially unsafe construct is only used in safe contexts throughout a particular program.

The Hindley-Milner type system (Hindley 1969; Milner 1978) with the Damas-Milner inference algorithm (Damas and Milner 1982) is a great historical achievement, because it is an expressive type system which allows to type day-to-day programs without any problems, and not only has an efficient type checking algorithm, but even permits efficient type inference! Thus, the programmer is not forced to annotate a program with types or declare the types of entities used – everything is inferred by the compiler automatically and checked for consistency. One of the highlights of this type system is the possibility for *parametrically polymorphic functions*. Functions that work in the same way for all datatypes need not be instantiated over and over again, making a new copy for each new datatype, but can be written once and used everywhere.

Haskell (Peyton Jones 2003), along with other languages such as SML (Milner *et al.* 1997) or Clean (Plasmeijer and van Eekelen 2001), are based on the Hindley-

Milner type checking rules. Haskell in particular has also proved to be a testbed for various type system extensions. Even in its standard form, Haskell has several extensions over the classic Hindley-Milner algorithm. It allows explicit type signatures, if desired, to restrict a function's type. It provides type classes, which allow one to write overloaded functions, with different functionality for different types. There is a kind system, which mirrors the type system yet again on the level of datatypes, and thus allows the systematic treatment of type constructors such as lists or tuples, which are parametrized over other types – and there is more . . .

However, Haskell – just as the other languages mentioned – suffers from a problem: There is a conflict between the use of static types to prevent type errors and the goal of code reuse. Static types, in particular *nominal* types (types distinguished by name rather than by structure) make a difference where there would not normally be one for the computer. For instance, a date is nothing more than a triple of integers, but in a static type system, a date will be treated as a separate type, and only special functions, tailored for dates, can be used on them.

Often, this is exactly what is desired, but sometimes, it is a hindrance as well. Parametrically polymorphic functions make the situation bearable. Declaring new types is cheap, because a lot of functions, especially functions that process the structure of the data, work on any type. Everything can be put into a list, or a tree, selected, rearranged, and so forth.

Parametrically polymorphic functions, however, are only good for doing things that are completely independent of the actual values. The values are put into a box and cannot be touched from within a polymorphic function. Often, however, we would want to make use of the underlying *structure* of the type. Suppose we have hidden an identification number, a room number, and a year in three different datatypes. This is usually a good thing, because most likely it should be prevented that a year is suddenly used as a room number. Still, all three are in principle integers, and integers admit some operations that are not applicable to all datatypes. They can be added, compared, tested for equality, incremented, and much more. Such functionality can never be captured in a parametrically polymorphic function, because it is only applicable to integers and, maybe, other numeric types, but not to all types, and *definitely* not to all types in the same way!

Type classes can help here. A function can be overloaded to work on different datatypes in a different fashion. For instance, equality is already overloaded in Haskell, and so is addition or comparison. We can make identification numbers, room numbers, and years instances of the appropriate classes, and then use the functions. But, as soon as we define a new datatype that is just an integer, we have to write the instance declarations all over again. Similarly, when we add a new function that works perfectly on integers, we have to overload it and write instance declarations for all the datatypes again.

What we really need is a controlled way to forget the distinctions that the type system makes for a time and define functions that mainly care about the *structure* of a type. The possibility to define functions by analysis of the structure of datatypes is what we call *generic programming* in the context of this thesis.

Haskell in particular offers a limited mechanism to achieve some of the desired functionality: the **deriving** construct allows automatic generation of type class instances for a fixed set of Haskell type classes – type classes that provide methods such as the equality test, comparison, lower and upper bounds of values, generating a canonical representation of values as a string, and reading such a representation back in. Thus, Haskell has some built-in generic programs, but does not allow you to write your own generic programs. Other generic functions, or variations of the functions that can be derived, have to be defined for each datatype by hand.

1.2 History of Generic Haskell and contributions of this thesis

In this section, we present a brief summary of how Generic Haskell came to life, and which language(s) it refers to. In this context, it will also become clear on whose work this thesis is based and what the contributions of this thesis are.

The first generic programming language extension that has been designed for Haskell is PolyP (Jansson and Jeuring 1997; Jansson 2000). In PolyP, generic functions are called *polytypic*. The language introduces a special construct in which such polytypic functions can be defined via structural induction over the structure of the pattern functor of a regular datatype. Regular datatypes in PolyP are a subset of Haskell datatypes. A regular datatype t must be of kind $* \to *$, and if a is the formal type argument in the definition, then all recursive calls to t must have the form $t\ a$. These restrictions rule out higher kinded datatypes as well as *nested* datatypes, where the recursive calls are of a different form.

In the lecture notes for a Summer School (Backhouse *et al.* 1999), theoretical background on *generic programming* is combined with an introduction to PolyP, thereby establishing generic programming as a synonym for polytypic programming in the context of Haskell.

Ralf Hinze reused the term *generic programming* for the ability to define type-indexed functions during his own presentation of a programming language extension for Haskell (Hinze 1999*a*, 2000*b*). As in PolyP, Haskell is extended with a construct to define type-indexed functions. Type indices can be of kind $*$ (for generic equality, or showing values) or $* \to *$ (for mapping functions or reductions), and in principle, for all kinds of the form $* \to \cdots \to *$. The approach has two essential advantages over PolyP: first, generic functions can be defined over the structure of datatypes themselves, without falling back to pattern functors. Second, nested types do not pose a problem for Hinze's theory. While most generic functions are easier to define over the structure of a datatype directly than via the pattern functor, some functions that make explicit use of the points of recursion, such as generic cata- and anamorphisms, become harder to define.

Hinze's approach still suffers from some limitations: for each kind, the type language over which generic functions are defined is a different one, thus making the extension difficult to implement. Furthermore, types of kinds that are not in the above-mentioned form are not allowed as type indices of generic functions. And even though the set of types over which a generic function is explicitly defined is variable, no types of complex kinds are allowed in that set.

These limitations are overcome in Hinze's later work (Hinze 2000*c*), where generic functions can be defined for datatypes of all kinds, using a single function definition. In other words, one generic function can not only be instantiated to type arguments of a fixed kind, but to type arguments of all kinds. The essence of the idea is captured in the paper's title: "Polytypic functions possess polykinded types". The type of the generic function (to be precise, the number of function arguments it takes) is determined by the kind of the datatype it is instantiated to.

In his "Habilitationsschrift" (Hinze 2000*a*), Hinze presents both his approaches in parallel, comparing them and favouring the second for an implementation which he calls "Generic Haskell". His thesis also contains some hints about how to implement an extension for Haskell, taking the peculiarities of the Haskell language into account. The translation of generic functions proceeds by specialization: specific instances of a generic function are generated for the types at which the function is used in the program. This has the advantage that type information can still be completely eliminated in the translation, allowing for an efficient translation. On the other hand, generic functions remain special constructs in the language which are not first-class: they cannot be passed as arguments to other functions.

In another paper (Hinze and Peyton Jones 2001), a possible extension for the Glasgow Haskell Compiler GHC (GHC Team) is proposed (which has also been implemented), that is integrated into the type class system of Haskell: generic functions are not available as separate declarations, but only as class methods. Haskell allows default definitions for class methods to be given, such that if an instance is defined for a class without a new definition for a certain method, then the default definition is used. Using "derivable type classes", generic default definitions are allowed, consisting of several cases for some basic types, thus allowing generic behaviour to be derived when a new instance is requested. In Haskell, type classes are parametrized by type arguments of fixed kinds. For this reason, the derivable type classes are more related to Hinze's first approach, and do not have kind-indexed types. Moreover, they *only* work for type indices of kind $*$. Several of the classes for which Haskell provides the **deriving** construct can now be defined generically. Other desirable classes, such as the Functor class which provides a generic mapping function, are still out of reach.

The first release of the Generic Haskell compiler (Clarke *et al.* 2001) – in the context of the "Generic Haskell" project at Utrecht University – was therefore separate from the GHC extension for derivable type classes, and supported type-indexed functions with kind-indexed types following Hinze's earlier suggestions (Hinze 2000*a*), without integrating genericity with the type class system.

The existence of the compiler made practical programming experiments possible, and these uncovered some weaknesses in expressivity which lead to the extensions described in the paper "Generic Haskell, specifically" (Clarke and Löh 2003): default cases, constructor cases, and generic abstraction. These extensions were incorporated into a second release (Clarke *et al.* 2002) of the Generic Haskell compiler, together with support for type-indexed datatypes (Hinze *et al.* 2002). Such generic datatypes mirror the principle of structural induction over the language of datatypes on the type level. In other words, datatypes can be defined which have an implementation that depends on the structure of a type argument.

While Hinze's framework provided a strong and solid foundation, it turned out to have some inherent weaknesses as well: the concept of kind-indexed types, which implies that different cases of a generic definition take a different number of arguments, proved to be difficult to explain in teaching, and unwieldy in presentations. The theory forces all generic functions into the shape of catamorphisms, where recursion is not explicit; instead, the recursive calls are passed as arguments to the function.

Transferred to ordinary Haskell functions, this means that we would need to write the factorial function as

$$\begin{aligned} &\mathit{fac}\ \ 0 &&= 1 \\ &\mathit{fac}\ \mathit{rec}\ (n+1) &&= (n+1) \cdot \mathit{rec}\ \ . \end{aligned}$$

Note that this is an example by analogy: the factorial function does not have to be written this way in Generic Haskell, only recursion on generic functions followed this principle. An additional argument *rec*, which is equivalent to the recursive call to *fac n*, is passed to the second case. Such a definition requires a clear understanding of how the mechanism works; it is not immediately obvious what is going on. It is much easier to make the recursion explicit:

$$\begin{aligned} &\mathit{fac}\ \ 0 &&= 1 \\ &\mathit{fac}\ \ (n+1) &&= (n+1) \cdot \mathit{fac}\ n\ \ . \end{aligned}$$

Furthermore, if recursion is only possible via explicitly passed arguments, it is also difficult to break free of the catamorphic scheme: what if we do not want to recurse on the immediate predecessor? Maybe we would rather call *fac* $(n - 1)$. Or what if we have two functions which are mutually recursive?

Dependency-style Generic Haskell (Löh *et al.* 2003) is an attempt to alleviate this problem by providing a clearer, more flexible syntax without losing any of the generality and the features that were previously available.

Dependencies also form the core of the presentation in this thesis. While the paper introduces dependencies as a modification of previous work on Generic Haskell, we present Generic Haskell from scratch, using dependencies from the very beginning. The main chapters dealing with dependencies are 5, 6, and 9. Because the introduction of dependencies addresses the foundations of Generic Haskell, it has implications on almost every aspect of it. Therefore, prior results have to be reevaluated in the new setting. We repeat fundamental aspects of Hinze's theory in Chapters 7, 10, and 11.

In later chapters, we concentrate on extensions to Generic Haskell that make it more expressive and easier to use. These results – as far as they have been published – are taken from the papers "Generic Haskell, specifically" (Clarke and Löh 2003) and "Type-indexed data types" (Hinze *et al.* 2002). This material is also adapted and revised to fit into the framework of Dependency-style Generic Haskell.

This thesis can be seen as a description of Generic Haskell in a consistent state after four years of research as well as an explanation of how to write a compiler for a language like Generic Haskell. In several places, we point out design decisions taken and sketch other possibilities to solve problems. The current state of the Generic Haskell implementation is described in Section 2.2.

A more detailed overview over the contents of this thesis is given in Section 1.4, after we have surveyed related work.

1.3 Related work on generic programming

First of all, it should be mentioned that *generic programming* is not an ideal term for the structural polymorphism that we talk about in this thesis, because the term is used by different communities in different meanings. Most notably, the object-oriented programming community, when talking about generic programming, mean about the same as is captured by parametric polymorphism in Hindley-Milner based type systems. Nevertheless, generic programming is the term that has been used for a while now (Bird *et al.* 1996) in the functional programming community to refer to structural polymorphism, i.e., functions defined over the structure of datatypes, and we will continue the habit.

Generic programming in functional languages has grown into a rich and diverse field, and it is hard to do justice to all the excellent work that has been done during the last years. We will pick a few examples which we think are especially related to the work presented in this thesis, knowing that we omit several noteworthy others.

1.3.1 Intensional type analysis

Using intensional type analysis (Harper and Morrisett 1995), it is possible to analyze types at runtime, for instance to select a more efficient implementation of a function. The idea is very similar to the type-indexed functions that we discuss in Chapter 4. Stephanie Weirich (2002) has extended intensional type analysis to higher-kinded type arguments, thereby following some of the ideas of Hinze and making the approach usable for generic programming as well, especially in a purely structural type system.

More recent work (Vytiniotis *et al.* 2004) throws some additional light on the relation between structural and nominal types. An intermediate language with support for both structural and nominal types and type-indexed functions which can be open (i.e., extensible with new cases) or closed is presented. The language does not provide an automatic transformation of datatypes into their underlying structure (cf. Chapters 10 and 17), but it could be used as a target language for Generic Haskell.

1.3.2 Scrap your boilerplate!

Recently, Ralf Lämmel and Simon Peyton Jones have joined forces to provide a workable generic programming extension directly in GHC. Two papers (Lämmel and Peyton Jones 2003, 2004) describe the extension. The fundamental idea is a different one than for Generic Haskell: genericity is created by extending a polymorphic, uniform traversal function with type-specific behaviour. For instance, the identity traversal can be extended with the function that increases all integers by 1, resulting in a function that still works for all datatypes, but is no longer parametrically polymorphic.

Internally, a type representation is passed for such functions, and a typesafe cast is used to check if special behaviour has been specified for a datatype. The strength of the approach lies in the manipulation of large data structures, which is related to what Generic Haskell can achieve by means of default cases – the relation is described further in Chapter 14. Furthermore, generic functions defined in the "boilerplate" style are first class and require no special treatment in the Haskell type system.

Even though the perspective of the approach is a different one, it can also be used to write many of the generic operations that work by structural induction over the language of types – such as equality, comparison, parsing and unparsing – which is the original motivation behind Generic Haskell.

However, the "boilerplate" approach does not support type-indexed types.

1.3.3 Template Haskell

Template Haskell (Sheard and Peyton Jones 2002) is a language extension of Haskell, allowing the programmer to write meta-programs that are executed at compile time. Meta-programs have access to the abstract syntax tree of the program, and can use the tree to perform reflection on already existing code and to produce new code.

Using Template Haskell, it is possible to write generic programs as meta-programs. At the call site of a generic function, a specialized version of the generic program that works for one specific type, can be spliced into the program code.

Template Haskell aims at being far more than just a generic programming extension, but in the area of generic programming, it suffers from a couple of disadvantages. Template Haskell does not yet have a type system, although there is ongoing work to resolve this problem (Lynagh 2004). While the code resulting from meta-programs is type checked normally by the Haskell compiler, there is no check that the template program can only produce correct code. Furthermore, the syntax trees that Template Haskell manipulates contain less information than one would need to write good generic programs. For instance, it is hard to detect recursion (for instance, between datatypes), and the syntax trees are not annotated with type information. Finally, Template Haskell does not support syntactic sugar for generic functions.

1.3.4 Generic Clean

The programming language Clean (Plasmeijer and van Eekelen 2001) provides a fully integrated generic programming facility (Alimarine and Plasmeijer 2001) that is based on the same ideas (Hinze 2000*c*) as Generic Haskell. The Clean extension is integrated into the type class system, which is very similar to Haskell's system of type classes. Generic functions are defined as special, kind-indexed classes: a few instances have to be defined, and others can then be derived automatically in a generic way.

Generic Clean does not allow for dependencies between type-indexed functions, which makes it difficult to write generic functions that use other generic functions on variable types.

Generic programming in Clean has been used for several applications, such as the generation of automatic tests (Koopman *et al.* 2003), in the context of dynamic types (Achten and Hinze 2002; Achten *et al.* 2003), and to generate components of graphical user interfaces (Achten *et al.* 2004). There has also been work on optimization of generic functions (Alimarine and Smetsers 2004).

1.3.5 Pattern calculus

The pattern calculus (Jay 2003), based on the constructor calculus (Jay 2001), provides a very flexible form of pattern matching that permits patterns of multiple types to occur in a single **case** construct. Furthermore, special patterns such as application patterns can be used to match against any constructor. Generic functions can be implemented by analyzing a value, using sufficiently general patterns. Jay's calculus allows type inference and has been implemented in the programming language FISH2.

Generic functions written in this system are first class. Still, the implementation does not rely on type information to drive the evaluation. Functions written in the pattern calculus are, however, more difficult to write: the set of patterns for which a function must be defined in order to behave generically is relatively large, and some of the patterns embody complicated concepts. Furthermore, because the pattern match works on a concrete value, not on a type, functions that produce values generically, such as parsers, are difficult to write.

1.3.6 Dependent types

Lennart Augustsson (1999) was the first to suggest the use of dependent types in the context of Haskell, and proposed a language called Cayenne. While type-indexed functions, such as provided by Generic Haskell, are functions (or values) that depend on a type argument, dependent types are types that depend on a value argument. However, by using dependent types one can simulate generic functions. In addition, dependent types allow several applications beyond type-indexed functions, but at the price of significantly complicating the type system.

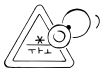
Using some features not provided by Cayenne, a number of complex encodings of generic functions within a dependently typed language have been presented by Altenkirch and McBride (2003). The style of dependent programming used in that paper is further developed in another article (McBride and McKinna 2004), and has led to the development of Epigram, another programming language that supports dependent types. Once the development has reached a stable point, it will be interesting to find out if generic programming can be provided in the form of a simple library in such a language, or if syntactic additions are still desirable to make generic programming practicable.

1.4 Selecting a route

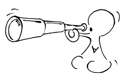
In Chapter 2, we make a remark about previous knowledge that we assume from readers, and discuss notational conventions.

We start our tour of Generic Haskell slowly, first introducing a core language in Chapter 3. This language consists of a subset of Haskell, and is held in a syntactic style that is mostly compatible with Haskell. Haskell can be relatively easily desugared to the core language. In this chapter, we also give type checking rules for the language and present a small step operational semantics.

In most of the other chapters, we will – step by step – introduce the features that make up Generic Haskell. We usually present some examples, both to motivate the need for the features that are presented, and to show how they can be used. For the examples, we usually use full Haskell syntax, to give an impression of what actual programs look like. After this leisurely introduction of a new feature, we generally discuss its implementation. We extend the core language with new constructs as necessary, and discuss the semantics of the new constructs, usually by presenting a translation back into the original core language. In these theoretical parts, we work exclusively on the core language and its extensions.



Often, such theoretical parts are marked by a "Lambda". The "Lambdas" are friendly fellows and experts on both functional and generic programming. They accompany the reader during his or her explorations. In this case, the "Lambda" is advising the reader who is interested more in practical usage of the language than in the gritty details, that the "Lambda"-marked section could be skipped without danger (at least on a first reading).



Similarly, on rare occasions, the "Lambda" warns that a certain area has not yet been explored in full detail, and that the following text describes future or ongoing work.

The chapters on additions to the core language are grouped into two parts. Chapters 4 to 11 introduce a basic language for generic programming, which is already quite powerful, but in many places not as convenient as would be desirable. Therefore, the remaining chapters focus on several extensions that allow writing generic programs more easily, but also aim at enhancing the expressiveness of the language even further.

Chapter 4 covers the first extension of the core language, the possibility to define type-indexed functions. Generic functions are type-indexed functions that fulfill specific conditions. It thus makes sense to introduce type-indexed functions first, in this chapter, and to discuss genericity later.

The idea of *dependencies* between type-indexed functions, as introduced in the paper "Dependency-style Generic Haskell" (Löh *et al.* 2003), forms the core of this thesis. Dependencies are first discussed in a limited setting in Chapter 5, which also slightly generalizes type-indexed functions by allowing more flexible type patterns. Chapter 6 gives a theoretical account of the extensions explained in Chapter 5. Only later, in Chapter 9, will we introduce dependencies in full generality.

In between, we sketch how generic functions work, and give the first few example generic functions, in Chapter 7. This is not new material, but covered in several of Hinze's papers on generic programming in Haskell.

In Chapter 8, we discuss the principle of local redefinition, which offers the possibility to locally modify the behaviour of a type-indexed function. Local redefinition forms an essential part of Dependency-style Generic Haskell.

After having presented the full story about dependencies in Chapter 9, we complete our account of generic functions in the Chapters 10, which focuses on how to map datatypes to a common structural representation, and 11, which explains how to translate generic functions, in particular if they are called on datatypes for which the behaviour has to be derived in a generic way. Both chapters are based on earlier work on Generic Haskell and adapted here such that they fit into our framework.

Chapter 11 at the same time marks the end of the central features of Generic Haskell. With the features discussed up to this point, one has a fully operational language at hand. Nevertheless, several extensions are extremely helpful, because they make generic programming both easier and more flexible. These additional features are introduced in the remainder of the thesis.

Generic abstraction, covered by Chapter 12, is another way of defining type-indexed functions. Functions defined by generic abstraction do not perform case analysis on a type argument directly, but use other type-indexed functions and inherit their type-indexed-ness from those. Generic abstraction is one of the extensions introduced in the "Generic Haskell, specifically" (Clarke and Löh 2003) paper. They are shown here from the perspective of Dependency-style, and benefit significantly from the changed setting, allowing for a far cleaner treatment than in the paper.

In Chapter 13, we discuss type inference, specifically for type-indexed functions, which can help to reduce the burden on the programmer while writing generic functions further. Several different problems are discussed, such as inference of the type arguments in calls to generic functions or the inference of type signatures of generic functions. While the answers to the questions about type inference for generic functions are not all positive, it is possible to infer a reasonable amount of information that allows comfortable programming. This chapter is for the most part ongoing work and based on unpublished material.

Chapter 14 on default cases presents a way to reuse cases from existing type-indexed functions while defining new generic functions. Frequently occurring traversal patterns can thus be captured in basic type-indexed functions, and subsequently extended to define several variations of that pattern. Default cases are covered in "Generic Haskell, specifically", but adapted for Dependency-style here.

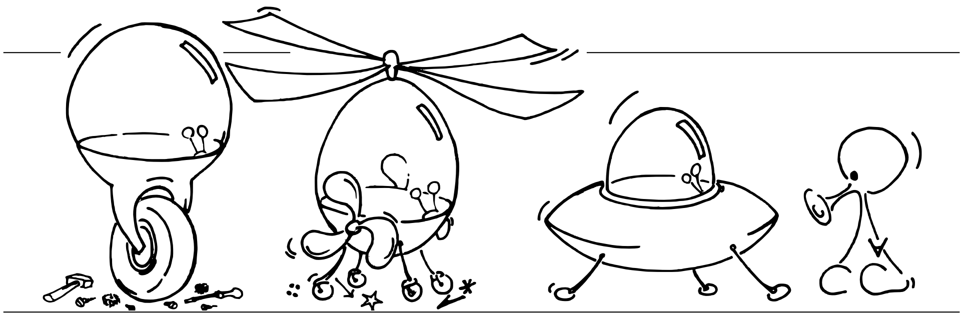
In Chapter 15, we extend our core language to allow all of Haskell's type declaration constructs: **type** and **newtype** as well as **data**. This becomes relevant in the following Chapter 16, on type-indexed datatypes, which is based on the paper of the same name (Hinze *et al.* 2002). Type-indexed datatypes are like type-indexed functions, but on the type level: they are datatypes that have a different implementation depending on the structure of a type argument. Again, the presentation of type-indexed datatypes is significantly different from the paper, because we extend the dependency type system to the type level.

In Chapter 17, we describe a number of different encodings of datatypes into the Haskell type language, which form alternatives to the standard encoding that is discussed in Chapter 10. By changing the view on datatypes, some generic functions become easier to define, others become more difficult or even impossible to write. This chapter describes ongoing work.

Chapter 18 on modules shows how type-indexed – and in particular generic – functions and datatypes can live in programs consisting of multiple modules, and to what extent separate compilation can be achieved. This chapter draws upon unpublished knowledge gained from the implementation of Generic Haskell (Clarke *et al.* 2002). Modules are also the last step in making Generic Haskell a complete language on top of Haskell; therefore this chapter concludes the thesis.

All in all, we present a complete and rich language for generic programming, which can, has been, and hopefully will be used for several interesting applications. I wish you a joyful exploration!

2 Choosing the Equipment



In this chapter, we will prepare for our exploration. Section 2.1 briefly discusses prerequisites for the material presented in this thesis and pointers to introductory material to gain the assumed knowledge. In Section 2.2, we discuss the status of the Generic Haskell compiler. In Section 2.3, we explain several notational conventions that are used in the remainder of this thesis.

2.1 Prerequisites

I have tried to write this thesis in such a way that it is understandable without a heavy background in generic programming in the context of functional languages. Because the thesis describes a language extension of Haskell, a familiarity with Haskell is very advisable. I recommend Bird's excellent textbook (Bird 1998), but the very readable language report (Peyton Jones 2003) might do as well, especially if another statically typed functional language, such as Clean or an ML variant, is already known. It is extremely helpful if one is comfortable with the **data** construct in Haskell to define new datatypes, and with the kind system that classifies types in the same way as types classify expressions.

In the formal discussion of the language, we use many deduction rules to define type checking algorithms and translations. It can therefore do no harm if one has seen such concepts before. I heartily recommend Pierce's book on "Types and programming languages" (Pierce 2002), which serves as a well-motivated general introduction into the theory of statically typed languages.

If one is interested in the background of generic programming beyond Generic Haskell, I recommend the Summer School article of Backhouse *et al.* (1999) – which contains an introduction to the field from a theoretical perspective, without focus on a specific programming language – or some of the materials mentioned in Section 1.3 on related work. If more examples or even exercises are desired, then the later Summer School articles of Hinze and Jeuring (2003*a*,*b*) provide material.

2.2 The Generic Haskell compiler

There exists a Generic Haskell compiler. There have been two formal releases, Amber (Clarke *et al.* 2001) and Beryl (Clarke *et al.* 2002), and the current development version supports a couple of features that the two released versions do not. Nevertheless, the compiler lags considerably behind the development of the theory. The compiler translates into Haskell, and leaves all type checking to the Haskell compiler. Furthermore, the support for Dependency-style Generic Haskell, as it is used in this thesis, is rudimentary at best, and type signatures have to be written using kind-indexed types (cf. Section 6.6). The syntax used in the compiler is discussed in the User's Guides for the two releases, and the Summer School material (Hinze and Jeuring 2003*a*,*b*) contains examples and exercises that are specifically targeted at the language that is implemented by the compiler.

Even with its limitations, the compiler can be used to implement a large number of the examples in this thesis and provide valuable insight into the practice of generic programming. The current version is available from `http://www.generic-haskell.org/`. It is my hope that in the future this site will have a version with full support for the features discussed in this thesis.

2.3 A note on notation

This section lists several notational conventions that we adhere to in this thesis.

2.3.1 Natural numbers and sequences

Natural numbers start with 0. Whenever the domain of a numerical value is not explicitly given, it is a natural number. We use $m\,..\,n$ to denote the set of natural numbers between m and n, including both m and n. We use $m\,..$ to denote the set of natural numbers greater than or equal to m, and $..\,n$ to denote the set of natural numbers smaller than or equal to n.

2.3.2 Syntactic equality

We use the symbol $\equiv$ to denote syntactic equality or meta-equality. The symbol $=$ is used in Haskell and our languages for declarations, and $==$ is used as a name for the equality function in Haskell and our languages.

2.3.3 Repetition

Whenever possible, the ellipsis $\ldots$ is *not* used in this thesis. Instead, we use repetition constructs that are defined as follows:

$$\begin{aligned}
\{X\}^{i \in m..n} &\mid m > n &&\equiv \varepsilon \\
&\mid \textit{otherwise} &&\equiv X[i \mathbin{/} m]\ \{X\}^{i \in m+1..n} \\
\{X\}_s^{i \in m..n} &\mid m > n &&\equiv \varepsilon \\
&\mid \textit{otherwise} &&\equiv X[i \mathbin{/} m]\ \{s\ X\}^{i \in m+1..n}
\end{aligned}$$

Read these definitions as syntactical macros where X and s represent arbitrary syntactical content, i is a variable, and m and n are natural numbers. The substitution is syntactical: where the bound variable i *literally* occurs in X, the natural number m is substituted.

Note that if X or s contain infix operators, the intention is that the associativity of the infix operator is interpreted *after* the expansion. For example, the list $1 : 2 : 3 : 4 : 5 : [\,]$, to be read as $1 : (2 : (3 : (4 : (5 : [\,]))))$, could be written as

$$\{i :\}^{i \in 1..5}\ [\,]\ .$$

Using Haskell's syntactic sugar for lists, we could write

$$[\{i\}_{,}^{i \in 1..5}]$$

for the same list, namely $[1, 2, 3, 4, 5]$.

2.3.4 Environments

Environments are finite maps, associating keys with values. For the entries we use different syntax depending on what an entry is meant to express. For demonstration, we will use entries of the form $x \mapsto v$, mapping a key x to a value v. Most often, we treat environments as lists that extend to the right:

Environments	
$\mathrm{E} ::= \varepsilon$	empty environment
$\quad \mid \ \mathrm{E}, x \mapsto v$	non-empty environment .

If not otherwise stated, an environment can only contain one binding for a certain key. If a second binding is added to the right, then it overwrites the old. Environments can be reordered. Consequently, we use the notation $x \mapsto v \in \mathrm{E}$, but also $\mathrm{E} \equiv \mathrm{E}', x \mapsto v$, to express the fact that the binding $x \mapsto v$ is contained in the environment E. We use the comma (,) also as union operator for two environments, again with the rightmost binding for a key overwriting all earlier bindings for the same key. We use the same notation for sets as for environments. Sets are environments with only keys, i.e., empty values.

2.3.5 Syntax descriptions, metavariables and indices

We introduce a multitude of languages in this thesis, and an important part of languages is syntax. The syntax of environments above is an example of how we present syntax descriptions: each syntactic category is named before it is listed, and each alternative is followed by a description to the right. The symbols on the right hand side of productions are either **terminals** that are part of the syntax (such as ε above), or metavariables that refer to **nonterminals** of different categories. The left-hand side of a production introduces the symbol that we use as metavariable for the category that is just defined, in this case E for environments.

In most cases, we use only one symbol to denote all occurrences of this syntactic category, and distinguish different entities by indexing. Thus, if E is an environment, then so are E', E_2, E_i, or E'_j. A list of all metavariables that we use and their meaning is shown on page 307.

Often, we use variable repetition constructs in syntax descriptions. For example, we could also have described the syntax of environments non-inductively using the following production:

Environments, alternatively

$\mathrm{E} ::= \{x_i \mapsto v_i\}_{,}^{i \in 1..n} \quad (n \in 0\,..)$

environment .

Here, we state that an environment may be any comma-separated sequence of key-value pairs of the form $x_i \mapsto v_i$, where that sequence may be of any length n in the given range. In the spirit of Section 2.3.1, we drop the explicit range specification $(n \in 0\,..)$ if the range is $0\,..$, i.e., unrestricted.

2.3.6 Deduction rules

Most properties and algorithms in this thesis are specified by means of deduction rules. A sample deduction rule is shown in Figure 2.1.

$$\mathrm{K};\Gamma \vdash e :: t$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 :: t_1 \to t_2 \\ \Gamma \vdash e_2 :: t_1\end{array}}{\Gamma \vdash (e_1\ e_2) :: t_2}\quad \text{(e-app-sample)}$$

Figure 2.1: Sample deduction rule

We give the form of the judgment before the rules, between horizontal lines. In this case, the judgment is of the form

$$\mathrm{K};\Gamma \vdash e :: t\ .$$

We use the same metavariables as in the syntactic descriptions, and distinguish multiple occurrences of objects of the same category by use of indices.

Each rule has a name, such as (e-app-sample) in the example. If an environment such as K in this case is passed on unchanged, we sometimes drop it from the rule.

2.3.7 Free variables and substitutions

We define which variables are free when we describe a particular language. Throughout this thesis, we use $\mathsf{fev}(e)$ to refer to free variables of an expression e, whereas $\mathsf{ftv}(t)$ refers to free type variables of a type t, and $\mathsf{fdv}(t)$ refers to free dependency variables (see Chapter 6) of a type t.

We write $e_1[e_2\ /\ x]$ to denote the application of the substitution that replaces every free occurrence of variable x in expression e_1 by expression e_2. We use substitutions also on other syntactic categories, for instance types. We use the metavariables φ and ψ to stand for substitutions, and then write $\varphi\ e$ to denote the application of a substitution φ to an expression e.

We usually assume that the application of substitutions does not lead to name capture, i.e., that alpha-conversion is performed in such a way that no name capture occurs.

2.3.8 Font conventions

We use bold face in the text to denote language **keywords** and for definitions. In the index, pages containing the definition of a certain notion are also in bold face.

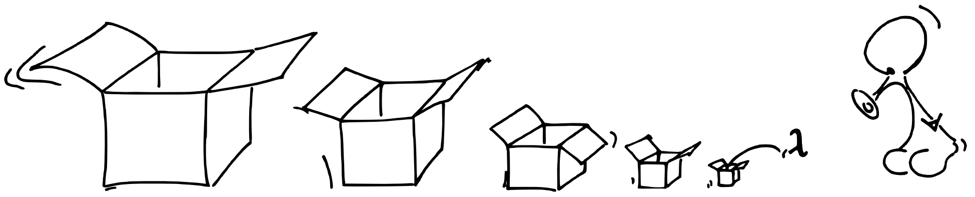
We emphasize *important* notions that are not defined elsewhere or not defined in this thesis. Again, we point to pages containing such notions using an emphasized page number in the index.

We use capital Greek letters to denote environments. Note that capital Greek letters are always written upright, so E denotes a capital "epsilon", whereas E is a capital "e" – we try to avoid such clashes as much as possible, though, to avoid confusion.

In code examples, next to the bold **keywords**, *identifiers* are written in italics, Type names are capitalized, *Constructors* capitalized in italics, and type $\mathsf{Classes}$ capitalized using a sans-serif font.

We also use a sans-serif font for internal operations, i.e., functions on the meta-language level, such as $\mathsf{fev}(e)$ to determine the free variables of an expression. The arguments of internal operations are always enclosed in parentheses.

3 A Functional Core



Before we will start on our tour into the field of generic functions, we will stick to the known for a while. While we will describe generic programming in the context of Generic Haskell, an extension to the Haskell language, in an informal way, we also need a vehicle for formal analysis of the generic features, a solid core language to base our extensions on. To this end, we will introduce a basic functional core language, named FC, in this chapter, much like what the Haskell Language Report (Peyton Jones 2003) uses as the target of translation for all of the advanced constructs defined in the Haskell language.

The language FC is designed to be sufficiently rich that, in conjunction with the gradual extensions following in future chapters, it can be used to demonstrate the full power of generic programming in a formal setting.

Although there are no language constructs specific to generic functions in this chapter and the language may look very familiar, this chapter also introduces some notations and mechanisms that will be used extensively throughout the rest of the thesis. This chapter is probably not necessary to understand the rest of the thesis, but is a good reference to look up unfamiliar notation or terminology.

Type systems and their properties for very similar languages are given in Pierce's book (2002).

3.1 Syntax of the core language FC

The syntax of the core language is shown in Figure 3.1.

A program is a list of datatype declarations followed by a special expression called **main**.

Datatype declarations are modelled after Haskell's **data** construct. A datatype may be parametrized (we write the type arguments using a type-level Λ, which may only occur in a datatype definition). A datatype has zero or more **constructors**, each of which has a number of arguments (or **fields**).

Kinds are the types of types. Kind $*$ is reserved for all types that can be assigned to expressions, whereas parametrized datatypes (also called type constructors) have functional kinds.

The type language has variables and **named types** (types that have been defined as datatypes). Two types can be applied to each other. We can universally quantify over a type variable. Function types are not explicitly included in the syntax, but we assume that the function type constructor $(\rightarrow)$ is part of the named types T, and available as a built-in type constant.

In expressions, one can refer to variables and constructors. Application and lambda abstraction deal with functions. An expression can be analyzed in a case statement. We call this expression the **head** of that case statement.

A case statement consists of multiple **arms**, where each arm consists of an expression that is **guarded** by a pattern. Patterns are a restricted form of expressions, consisting of (fully applied) constructors and variables only. All variables in a pattern must be distinct.

A let statement allows to locally define a new function via a function declaration. The value bound in the let is visible in the expression that constitutes the **body** of the let. It is, however, not visible in the definition of the local value itself, i.e., this is *not* a recursive let. The **fix** statement can be used to introduce recursion.

Using **fix** and assuming that we have n-ary tuples for arbitrary n, we can define a recursive let statement **letrec** as a derived form. In Section 3.6, we will formally introduce recursive let as a language extension, and subsequently replace **let** with **letrec**, as in Haskell.

We will often assume that certain datatypes, such as tuples and lists or integers, and some primitive functions on them, are predefined. We will also often use more convenient notation for some operations and constructs, such as the tuple and list notation that is standard in Haskell, grouping of multiple lambda abstractions, and infix operators. In particular (and probably most important), we will write

$$t_1 \rightarrow t_2$$

Programs		
P	$::= \{D_i;\}^{i\in 1..n}\ \mathbf{main} = e$	type declarations plus main expression
Type declarations		
D	$::= \mathbf{data}\ T = \{\Lambda a_i :: \kappa_i.\}^{i\in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{\mid}^{j\in 1..m}$	datatype declaration
Value declarations		
d	$::= x = e$	function declaration
Kinds		
κ	$::= *$	kind of manifest types
	$\mid\ \kappa_1 \to \kappa_2$	functional kind
Types		
t, u	$::= a, b, c, f, \ldots$	type variable
	$\mid\ T$	named type
	$\mid\ (t_1\ t_2)$	type application
	$\mid\ \forall a :: \kappa.t$	universal quantification
Expressions		
e	$::= x, y, z, \ldots$	variable
	$\mid\ C$	constructor
	$\mid\ (e_1\ e_2)$	application
	$\mid\ \lambda x \to e$	lambda abstraction
	$\mid\ \mathbf{case}\ e_0\ \mathbf{of}\ \{p_i \to e_i\}_{;}^{i\in 1..n}$	case
	$\mid\ \mathbf{let}\ d\ \mathbf{in}\ e$	let
	$\mid\ \mathbf{fix}\ e$	fixed point
Patterns		
p	$::= (C\ \{p_i\}^{i\in 1..n})$	constructor pattern
	$\mid\ x, y, z, \ldots$	variable pattern

Figure 3.1: Syntax of the core language FC

for the type of functions from t_1 to t_2. All these constructs are used to make examples more readable. It is easy to translate them away.

In the following, we will define when an FC program is correct and present an operational semantics of the language.

3.2 Scoping and free variables

The language FC has the usual scoping rules. Type variables are bound only by type-level lambda Λ in datatype declarations, and by universal quantifiers. Abstracted type variables in datatypes are visible in the entire datatype definition; quantified type variables everywhere underneath the quantifier. We use $\mathsf{ftv}(t)$ to refer to the free type variables of a type t.

Constructor and type names scope over the whole program.

Value-level variables are introduced by a let statement, by a lambda abstraction, and by the patterns in a case statement. A let binds all variables that occur on the left hand side of its declarations in its body. A lambda abstraction binds its variable in its body, too. A pattern binds all variables that occur in the pattern in the expression that corresponds to the pattern. Patterns are only legal if all variables in one pattern are different. We use $\mathsf{fev}(e)$ to refer to the free variables of an expression e.

Programs are equivalent under alpha-conversion, i.e., bound variables can be renamed without changing the meaning of the program.

Substitution is generally meant to be capture-avoiding: if the substituted expression contains variables that would be captured by a binding construct, it is assumed that alpha-conversion is performed so that no name capture takes place. If name capture is intended, we will explicitly mention that fact.

3.3 Types and kinds

For the large part of this thesis, we will not consider type *inference* (an exception is Chapter 13). Nevertheless, type *safety* is an essential part of generic programming and thus of Generic Haskell. Therefore we present rules that specify when a program in the core language can be assigned a type. We will not, however, present algorithms to find a suitable type.

We annotate all type variables with kinds; therefore finding the kind of a type is straightforward, given the kind rules that follow. On the value level, however, we do not use type annotations explicitly, but assume that they are given as needed, in addition to the program.

One way to view the core language is as a variant of the polymorphic lambda calculus F_ω (Girard 1972), where all the type applications and type abstractions have been separated from the program, but can be recovered during the type checking phase as needed. Leaving out the type annotations allows us to focus on the part that is central to this thesis – the generic programming – and also to write the examples in a language which is closer to what the programmer will actually write in a real programming language, which will be able to infer types at least for expressions that Haskell can infer types for.

Our language is more expressive than Haskell, though. We allow universal quantification to occur everywhere in types, thereby opening the possibility to express types of arbitrary rank, as opposed to Haskell 98, which only admits rank-1 types. Rank-n types do occur frequently in the treatment of generic functions, and it is desirable that a language for generic programming supports them. They are implemented as an extension to Haskell and useful in many other areas (Peyton Jones and Shields 2003).

Furthermore, we allow universally quantified types to occur within datatypes, and even datatypes parametrized with polymorphic types, but these features are rarely used and not essential.

As datatypes play a central role in Generic Haskell, it is important to be familiar with the concept of kinds. Let us therefore review the Haskell kind system.

The kind $*$ is reserved for types that can be assigned to expressions in the language, whereas parametrized types (or **type constructors**) behave like functions on the type level and are assigned functional kinds. For instance, the Haskell types Int, Char and Bool are all of kind $*$, whereas Maybe, the list constructor [ ], or the input-output monad IO are of kind $* \to *$. The pair constructor (,) and Either are both of kind $* \to * \to *$. We will sometimes abbreviate $* \to * \to *$ to $*^2$, where

$$*^n = \{* \to\}^{i \in 1..n} * \ .$$

Kinds need not be of the form $*^n$ for some natural number n, as types can be parametrized over type constructors. A simple example is

$$\textbf{data}\ \mathrm{IntStruct}\ (f :: * \to *) = I\ (f\ \mathrm{Int})\ .$$

Here, f is a variable of kind $* \to *$ that can be instantiated, for example, to some container type. Hence, IntStruct has kind $(* \to *) \to *$. We will encounter more examples of types with complex kinds later.

During kind checking, we make use of an environment K that contains bindings of the form $a :: \kappa$ and $T :: \kappa$, associating type variables and named types with kinds. The kind checking rules are given in Figure 3.2 and are of the form

$$\mathrm{K} \vdash t :: \kappa\ ,$$

$$\mathrm{K} \vdash t :: \kappa$$

$$\frac{a :: \kappa \in \mathrm{K}}{\mathrm{K} \vdash a :: \kappa}\ \text{(t-var)} \qquad \frac{T :: \kappa \in \mathrm{K}}{\mathrm{K} \vdash T :: \kappa}\ \text{(t-named)}$$

$$\frac{\mathrm{K} \vdash t_1 :: \kappa_1 \to \kappa_2 \quad \mathrm{K} \vdash t_2 :: \kappa_1}{\mathrm{K} \vdash (t_1\ t_2) :: \kappa_2}\ \text{(t-app)}$$

$$\frac{\mathrm{K}, a :: \kappa \vdash t :: *}{\mathrm{K} \vdash \forall a :: \kappa.\ t :: *}\ \text{(t-forall)}$$

Figure 3.2: Kind checking for core language of Figure 3.1

expressing that under environment K, the type t has kind κ. The rules themselves bear no surprises: variables and named types are looked up in the environment, application eliminates functional kinds, and universal quantification works for variables of any kind, but the resulting type is always of kind $*$.

The type rules that are displayed in Figure 3.3 are of the form

$$\mathrm{K}; \Gamma \vdash e :: t\ ,$$

which means that expression e is of type t under environment Γ and kind environment K. The environment K is of the same form as in the kind inference rules, whereas Γ contains entries of the form $x :: t$ and $C :: t$, associating a type t with a variable x or a constructor C. Recall our convention from Section 2.3.6 that we may omit environments that are not important to a rule (i.e., passed on unchanged). The kind environment K is rarely needed during type checking.

Again, most rules are simple: variables and constructors are looked up in the environment. Application and lambda abstraction are simple. Also, let statements are easy to check as they are non-recursive.

The rule for case statements is the most complicated rule: The head of the case, e_0, must type check against a type t_0. This type must match the type of the patterns. Each pattern can bind a number of variables, which is witnessed by the environments Γ_i. Each arm e_i is now checked with an extended environment where the variables bound by the respective pattern are added. All arms must be of the same type t, which is then also the type of the whole expression. The rules for pattern matching are shown in Figure 3.4 and explained in Section 3.3.1.

$$\mathrm{K}; \Gamma \vdash e :: t$$

$$\frac{x :: t \in \Gamma}{\Gamma \vdash x :: t}\ \text{(e-var)} \qquad \frac{C :: t \in \Gamma}{\Gamma \vdash C :: t}\ \text{(e-con)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 :: t_1 \to t_2 \\ \Gamma \vdash e_2 :: t_1\end{array}}{\Gamma \vdash (e_1\ e_2) :: t_2}\ \text{(e-app)} \qquad \frac{\begin{array}{c}\mathrm{K} \vdash t_1 :: * \\ \mathrm{K}; \Gamma, x :: t_1 \vdash e :: t_2\end{array}}{\mathrm{K}; \Gamma \vdash \lambda x \to e :: t_1 \to t_2}\ \text{(e-lam)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 :: t_0 \\ \{\Gamma \vdash^{pat} p_i :: t_0 \rightsquigarrow \Gamma_i\}^{i \in 1..n} \\ \mathrm{K} \vdash t :: * \\ \{\mathrm{K}; \Gamma, \Gamma_i \vdash e_i :: t\}^{i \in 1..n}\end{array}}{\mathrm{K}; \Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\ \{p_i \to e_i\}_{;}^{i \in 1..n} :: t}\ \text{(e-case)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e' :: t' \\ \Gamma, x :: t' \vdash e :: t\end{array}}{\Gamma \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e :: t}\ \text{(e-let-val)} \qquad \frac{\Gamma \vdash e :: t \to t}{\Gamma \vdash \mathbf{fix}\ e :: t}\ \text{(e-fix)}$$

$$\frac{\begin{array}{c}a \notin \mathsf{ftv}(\Gamma) \\ \mathrm{K}, a :: \kappa; \Gamma \vdash e :: t\end{array}}{\mathrm{K}; \Gamma \vdash e :: \forall a :: \kappa.\ t}\ \text{(e-gen)} \qquad \frac{\begin{array}{c}\Gamma \vdash e :: t_1 \\ \vdash t_1 \leqslant t_2\end{array}}{\Gamma \vdash e :: t_2}\ \text{(e-subs)}$$

Figure 3.3: Type checking for core language of Figure 3.1

The fix behaves as a fixpoint construct. Therefore the expression must have the type of a function where domain and codomain are both of type t. The least fixpoint is then of type t.

A type can be generalized, that is, universally quantified with respect to a type variable, if it can be type checked without any assumption about the type variable in the type environment Γ.

Finally, there is a rule for subsumption. Polymorphic types can be specialized, and monomorphic types generalized under certain circumstances. The subsumption relation is shown in Figure 3.5 and described in Section 3.3.2.

$$\Gamma_1 \vdash^{pat} p :: t \rightsquigarrow \Gamma_2$$

$$\frac{}{\Gamma \vdash^{pat} x :: t \rightsquigarrow x :: t} \quad \text{(p-var)}$$

$$\frac{C :: \{t_i \rightarrow\}^{i \in 1..n}\ t_0 \in \Gamma \qquad \{\Gamma \vdash^{pat} p_i :: t_i \rightsquigarrow \Gamma_i\}^{i \in 1..n}}{\Gamma \vdash^{pat} C\ \{p_i\}^{i \in 1..n} :: t_0 \rightsquigarrow \{\Gamma_i\}_{,}^{i \in 1..n}} \quad \text{(p-con)}$$

Figure 3.4: Type checking for patterns, extends Figure 3.3

$$\mathrm{K} \vdash t_1 \leqslant t_2$$

$$\frac{}{\vdash t \leqslant t} \quad \text{(s-refl)} \qquad \frac{\vdash t_3 \leqslant t_1 \qquad \vdash t_2 \leqslant t_4}{\vdash (t_1 \rightarrow t_2) \leqslant (t_3 \rightarrow t_4)} \quad \text{(s-fun)}$$

$$\frac{a \notin \mathsf{ftv}(t_1) \qquad \mathrm{K}, a :: \kappa \vdash t_1 \leqslant t_2}{\mathrm{K} \vdash t_1 \leqslant \forall a :: \kappa.\ t_2} \quad \text{(s-skol)} \qquad \frac{\mathrm{K} \vdash u :: \kappa \qquad \mathrm{K} \vdash t_1[u \mathbin{/} a] \leqslant t_2}{\mathrm{K} \vdash \forall a :: \kappa.\ t_1 \leqslant t_2} \quad \text{(s-inst)}$$

Figure 3.5: Subsumption relation on core types, extends Figure 3.3

3.3.1 Patterns

For patterns, we use a variation of the typing judgment that tells us about the variables that are bound by the pattern. The form of the judgment is

$$\Gamma_1 \vdash^{pat} p :: t \rightsquigarrow \Gamma_2 \ ,$$

which means that under environment Γ_1 the pattern p is of type t, binding the environment Γ_2.

A variable can have any type, and the variable is bound to that type. Note that pattern variables are not taken from the environment: patterns *bind* variables rather than *refer* to them.

A constructor pattern is checked as follows: the constructor has to be in the environment. In the pattern, the constructor has to appear fully applied, i.e., if its type is a function with n arguments, then there must be n argument patterns p_i of matching type. The pattern binds all variables bound by the arguments. We require all the bound variables to be distinct.

3.3.2 Subsumption

The subsumption judgments are of the form

$$\mathrm{K} \vdash t_1 \leqslant t_2 \ ,$$

which expresses the fact that an expression of type t_1 can behave as an expression of type t_2 under kind environment K.

The relation is reflexive and furthermore has two rules that deal with adding and removing universal quantifiers. The function type constructor is contravariant in its first argument, which is reflected in the subsumption rule for functions.

3.4 Well-formed programs

Some loose ends remain to be tied: for example, constructors need to be present in the environments, but are nowhere inserted. In fact, constructors are defined by the data declarations.

Figure 3.6 shows how: each type that occurs in one of the constructors (i.e., all the $t_{j,k}$'s) must be of kind $*$, where the kind environment is extended with the type variables that constitute the parameters of the datatype. For each constructor, a type is added to the resulting environment.

A whole program can now be checked as shown in Figure 3.7: checking proceeds against two environments K_0 and Γ_0, which may contain *external* types and functions. We always assume that at least $(\to) :: * \to * \to *$ is in K_0, but we often add additional types and functions on them, such as tuples or integers and arithmetic operations. All datatype declarations are checked for well-formedness against an extended environment K, which consists of K_0 plus all the datatypes declared in the program with their kinds. This expresses the fact that all datatypes may be mutually recursive. The Γ_i containing the constructors are added to Γ_0 to form Γ, and using this Γ, the main function is type checked. The type of the main function is also the type of the program.

3.5 Operational semantics

We define **values**, which are intended to be results of programs, in Figure 3.8. Values are a subset of the expression language, built from constructors (constructors are distinguished from functions by the fact that they cannot be reduced) and lambda abstractions. In addition, there is **fail**, which represents run-time failure. Failure will be used as the result of a case expression where none of the patterns matches the expression in the head.

Heads of case expressions are evaluated to weak head-normal form to reveal the top-level constructor during pattern matching. The syntax for weak head-normal form expressions is like the syntax for values, only that the arguments of the top-level constructor do not have to be evaluated and can be arbitrary expressions.

We extend the expression language with failure as well, as failure can occur while reducing expressions. The new type rule for **fail** is given in Figure 3.9: as **fail** represents failure, it can have any type.

Figure 3.10 presents small-step reduction rules to reduce expressions to values. We write

$$\vdash e_1 \rightarrowtail e_2$$

to denote that e_1 reduces to e_2 in one reduction step.

The reduction of a program is equivalent to the reduction of the main expression. In other words, the datatype declarations are not needed for the reduction of an expression. Constructors are syntactically distinguished, and knowing that a name represents a constructor is enough to formulate the reduction rules.

The reduction rules can be extended by additional rules for built-in functions. If a program is type checked under an initial type environment Γ_0 containing primitive functions (not constructors), it is necessary to provide reduction rules for these functions as well.

$$\mathrm{K}_1 \vdash D \leadsto \mathrm{K}_2; \Gamma$$

$$\frac{\begin{array}{c} D \equiv \textbf{data}\ T = \{\Lambda a_i :: \kappa_i.\}^{i \in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m} \\ \{\{\mathrm{K}\ \{, a_i :: \kappa_i\}^{i \in 1..\ell} \vdash t_{j,k} :: *\}^{k \in 1..n_j}\}^{j \in 1..m} \\ \Gamma \equiv \{C_j :: \{\forall a_i :: \kappa_i.\}^{i \in 1..\ell}\ \{t_{j,k} \rightarrow\}^{k \in 1..n_j}\ T\ \{a_i\}^{i \in 1..\ell}\}_{,}^{j \in 1..m} \end{array}}{\mathrm{K} \vdash D \leadsto T :: \{\kappa_i \rightarrow\}^{i \in 1..\ell} *; \Gamma} \quad \text{(p-data)}$$

Figure 3.6: Well-formed data declarations

$$\mathrm{K}; \Gamma \vdash P :: t$$

$$\frac{\begin{array}{c} P \equiv \{D_i;\}^{i \in 1..n}\ \textbf{main} = e \\ \{\mathrm{K} \vdash D_i \leadsto \mathrm{K}_i; \Gamma_i\}^{i \in 1..n} \\ \mathrm{K} \equiv \mathrm{K}_0, \{\mathrm{K}_i\}_{,}^{i \in 1..n} \quad \Gamma \equiv \Gamma_0, \{\Gamma_i\}_{,}^{i \in 1..n} \\ \mathrm{K}; \Gamma \vdash e :: t \end{array}}{\mathrm{K}_0; \Gamma_0 \vdash P :: t} \quad \text{(p-prog)}$$

Figure 3.7: Well-formed programs

Values		
v	$::= C\ \{v_i\}^{i \in 1..n}$	constructor
	$\mid\ \lambda x \rightarrow e$	function
	$\mid\ \textbf{fail}$	run-time failure
Weak head-normal form		
w	$::= C\ \{e_i\}^{i \in 1..n}$	constructor
	$\mid\ \lambda x \rightarrow e$	function
	$\mid\ \textbf{fail}$	run-time failure
Expressions		
e	$::= \ldots$	everything from Figure 3.1
	$\mid\ \textbf{fail}$	run-time failure

Figure 3.8: Syntax of values, extends Figure 3.1

$$\mathrm{K}; \Gamma \vdash e :: t$$

$$\frac{\mathrm{K}; \Gamma \vdash t :: *}{\mathrm{K}; \Gamma \vdash \textbf{fail} :: t} \quad \text{(e-fail)}$$

Figure 3.9: Type rule for run-time failure, extends Figure 3.3

Let us look at the reduction rules in a little bit more detail. The rule (r-con) causes that, if a constructor is encountered, its arguments are reduced to values one by one. The rule for application implements lazy evaluation and beta reduction: the function is reduced before its argument, and when a lambda abstraction is encountered, the argument is substituted for the variable in the body of the function. Run-time failure is propagated.

Most involved is, once more, the treatment of case statements. We try to match the first pattern against the head expression. If the match succeeds, as in (r-case-1), the match results in a substitution φ (mapping the variables bound in the pattern to the matching parts of the expression). Note that alpha-conversion may need to be performed on the arm $p_1 \rightarrow e_1$ prior to the derivation of φ in order to avoid name capture. The substitution φ is then applied to the expression belonging to the first arm. If the match fails, as in (r-case-2), the first arm of the case is discarded, i.e., matching continues with the next arm. If no arms are available anymore, as in (r-case-3), there is nothing we can do but to admit run-time failure.

A let statement is reduced by substituting the expression for the variable in the body. Finally, a fixpoint statement introduces recursion.

We have not yet shown how to perform a pattern match. This is detailed in Figure 3.11. These rules have conclusions of the form

$$\vdash^{match} p \leftarrow e \rightsquigarrow \varphi \ .$$

This is to be understood as that pattern p matches value e, yielding a substitution φ, mapping variables to expressions. We assume that **fail** is a special substitution, mapping all variables to the expression **fail**.

First, the expression to be matched is reduced using rule (m-reduce) until it reaches weak head-normal form. We use the internal function $\mathsf{whnf}(e)$ to express the syntactic condition that e is in weak head-normal form. The rule (m-con) shows that if both pattern and expression are of the same constructor, the arguments are matched one by one. The resulting substitution is the reverse composition of all substitutions originating from the arguments. (The order of composition is actually not important, because the pattern variables do not occur in the substituting expressions and the domains of the substitutions will be disjoint, as we declare patterns legal only if each variable occurs at most once.) Note that if one of the substitutions is **fail**, then the resulting substitution is **fail**, too. In rule (m-var), we see that matching against a variable always succeeds, binding the variable to the expression. The other rules indicate that everything else fails: matching two different constructors against each other, or matching a function against a constructor, and matching the **fail** value against anything.

There is a slight deviation from Haskell here, which simplifies the rules a little, but is otherwise nonessential: if the evaluation of an expression during a pattern match fails, then only this particular match fails, but the entire case statement may still succeed.

We can capture the safety of the reduction relation in the following two theorems. The progress and preservation properties are easy to prove for this language, as it is a core language without any special features. Readers familiar with such proofs may want to skip ahead to the next section.

Theorem 3.1 (Progress). If e is a closed expression (i.e., $\mathsf{fev}(e) \equiv \varepsilon$) and $\mathrm{K};\Gamma \vdash e :: t$, then either e is a value or there is an e' with $\vdash e \rightarrowtail e'$.

Note that there is no guarantee of termination, as we have unbounded recursion by means of the fix statement.

Proof. We prove the theorem by induction on the type derivation for e. The last rule applied cannot have been (e-var), because e is closed. If it was (e-con), then e is a constructor and thereby a value. Likewise for (e-lam) and (e-fail).

$$\vdash e_1 \rightarrowtail e_2$$

$$\frac{n \in 1.. \qquad \vdash e_1 \rightarrowtail e_1'}{\vdash C\ \{v_i\}^{i\in 1..m}\ \{e_i\}^{i\in 1..n} \rightarrowtail C\ \{v_i\}^{i\in 1..m}\ e_1'\ \{e_i\}^{i\in 2..n}}\ \text{(r-con)}$$

$$\frac{\vdash e_1 \rightarrowtail e_1'}{\vdash (e_1\ e_2) \rightarrowtail (e_1'\ e_2)}\ \text{(r-app-1)}$$

$$\frac{}{\vdash ((\lambda x \rightarrow e_1)\ e_2) \rightarrowtail e_1[e_2 \,/\, x]}\ \text{(r-app-2)}$$

$$\frac{}{\vdash (\mathbf{fail}\ e) \rightarrowtail \mathbf{fail}}\ \text{(r-app-3)}$$

$$\frac{n \in 1.. \qquad \vdash^{match} p_1 \leftarrow e \rightsquigarrow \varphi \qquad \varphi \not\equiv \mathbf{fail}}{\vdash \mathbf{case}\ e\ \mathbf{of}\ \{p_i \rightarrow e_i\}_{;}^{i\in 1..n} \rightarrowtail \varphi\ e_1}\ \text{(r-case-1)}$$

$$\frac{n \in 1.. \qquad \vdash^{match} p_1 \leftarrow e \rightsquigarrow \mathbf{fail}}{\vdash \mathbf{case}\ e\ \mathbf{of}\ \{p_i \rightarrow e_i\}_{;}^{i\in 1..n} \rightarrowtail \mathbf{case}\ e\ \mathbf{of}\ \{p_i \rightarrow e_i\}_{;}^{i\in 2..n}}\ \text{(r-case-2)}$$

$$\frac{}{\vdash \mathbf{case}\ e\ \mathbf{of}\ \varepsilon \rightarrowtail \mathbf{fail}}\ \text{(r-case-3)}$$

$$\frac{}{\vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightarrowtail e_2[e_1 \,/\, x]}\ \text{(r-let)} \qquad \frac{}{\vdash \mathbf{fix}\ e \rightarrowtail e\ (\mathbf{fix}\ e)}\ \text{(r-fix)}$$

Figure 3.10: Reduction rules for core language of Figure 3.1

$$\vdash^{match} p \leftarrow e \rightsquigarrow \varphi$$

$$\frac{\neg\mathsf{whnf}(e) \qquad \vdash e \rightarrowtail e' \qquad \vdash^{match} x \leftarrow e' \rightsquigarrow \varphi}{\vdash^{match} x \leftarrow e \rightsquigarrow \varphi} \quad \text{(m-reduce)}$$

$$\frac{\{\vdash^{match} p_i \leftarrow e_i \rightsquigarrow \varphi_i\}^{i \in 1..n} \qquad \varphi \equiv \{\varphi_{n+1-i}\}^{i \in 1..n}_{.}}{\vdash^{match} C\ \{p_i\}^{i \in 1..n} \leftarrow C\ \{e_i\}^{i \in 1..n} \rightsquigarrow \varphi} \quad \text{(m-con)}$$

$$\frac{}{\vdash^{match} x \leftarrow w \rightsquigarrow (x \mapsto w)} \quad \text{(m-var)}$$

$$\frac{C_1 \not\equiv C_2}{\vdash^{match} C_1\ \{p_i\}^{i \in 1..n} \leftarrow C_2\ \{e_j\}^{j \in 1..n} \rightsquigarrow \mathbf{fail}} \quad \text{(m-fail-1)}$$

$$\frac{}{\vdash^{match} C\ \{p_i\}^{i \in 1..n} \leftarrow \lambda x \rightarrow e \rightsquigarrow \mathbf{fail}} \quad \text{(m-fail-2)}$$

$$\frac{}{\vdash^{match} C\ \{p_i\}^{i \in 1..n} \leftarrow \mathbf{fail} \rightsquigarrow \mathbf{fail}} \quad \text{(m-fail-3)}$$

Figure 3.11: Pattern matching for the core language of Figure 3.1, extends Figure 3.10

In the case (e-app) (i.e., $e \equiv (e_1\ e_2)$), by induction hypothesis e_1 is either a value or can be reduced. If e_1 can be reduced, then (r-app-1) applies. If e_1 is a value, then e_1 is either **fail** or a lambda abstraction or a (partially applied) constructor. Depending on that, either (r-app-3) or (r-app-2) or (r-con) can be applied.

If the last type rule used is (e-case) and there is at least one arm in the statement, then (r-case-1) or (r-case-2) applies, depending on whether the pattern match succeeds or fails. We assume here that the pattern match can always be performed, which can be proved by another induction argument. Rule (r-case-3) is left for the case that there are no arms.

If the type derivation ends in a (e-let-val) or (e-fix), the corresponding reduction rules (r-let) and (r-fix) are possible.

For both (e-gen) and (e-subs), applying the induction hypothesis yields the proposition immediately. □

Theorem 3.2 (Type preservation). If $K; \Gamma \vdash e :: t$ and $\vdash e \rightarrowtail e'$, then $K; \Gamma \vdash e' :: t$.

This property, often called *subject reduction*, means in words that if an expression e can be assigned type t under some environment, and e reduces to e', then e' still has type t under the same environment.

Proof. Again, we proceed by induction on the typing derivation. If the last rule is (e-var) or (e-con), then there are no reduction rules that can be applied. The same holds for rules (e-lam) and (e-fail).

In the case that the last rule is (e-app), four reduction rules are possible: in the case of (r-app-3), the proposition trivially holds, and in the case of (r-app-1), the proposition follows immediately from the induction hypothesis applied to the function. The same holds for the case of (r-con); here, the induction hypothesis is applied to the first non-value argument of the constructor. If the reduction rule applied is (r-app-2), then we need the fact that a type is preserved under substitution, which we will prove as Lemma 3.3.

This lemma also helps us in case (e-let-val), where the only reduction rule that applies is (r-let).

If we have a case statement and the last type rule used is (e-case), there are three reduction rules that could apply: for (r-case-2), we see from the form of (e-case) that an arm from a case statement still allows the same type judgment as before. For (r-case-3), the proposition holds again trivially, which leaves (r-case-1). In this situation,

$$e \equiv \textbf{case}\ e_0\ \textbf{of}\ \{p_i \rightarrow e_i\}_{;}^{i \in 1..n} \quad .$$

Let $\{x_i\}_{,}^{i \in 1..m}$ be the variables contained in p_1. Then, by (e-case), we have

$$\Gamma, \Gamma_1 \vdash e_1 :: t$$

where $\Gamma_1 \equiv \{x_i :: t_i\}_{,}^{i \in 1..m}$ (we can prove this equality using a straightforward induction on the pattern ($\vdash^{pat}$) judgments). From (r-case-1), we know that

$$\vdash^{match} p_1 \leftarrow e \rightsquigarrow \varphi \,,$$

and using a similarly straightforward induction on the pattern matching ($\vdash^{match}$) judgments, we can show that φ is a composition of simple substitutions replacing each x_i with some expression of type t_i. Applying Lemma 3.3 then leads to the desired result that $\Gamma \vdash \varphi\, e_1 :: t$.

If the last step has been (e-fix), then $e \equiv \mathbf{fix}\; e_0$. Hence, we have to show that $\Gamma \vdash e_0\; (\mathbf{fix}\; e_0) :: t$. But $\Gamma \vdash e_0 :: t \rightarrow t$, and $\Gamma \vdash \mathbf{fix}\; e_0 :: t$, therefore an application of (e-app) does the trick.

In the case (e-gen), we have to show that $\vdash e' :: \forall a :: \kappa.\; t$. From the induction hypothesis, applied to $e :: t$, we know that $K, a :: \kappa; \Gamma \vdash e' :: t$. Because $a \notin \mathsf{ftv}(\Gamma)$, we can immediately reapply (e-gen) to get the desired result.

If the last derivation step is using rule (e-subs), then there is a type t' with $\vdash t' \leqslant t$ and $\vdash e :: t'$. We can apply the induction hypothesis to $e :: t'$, resulting in $\vdash e' :: t'$. Reapplying (e-subs), which only depends on the type, not on the expression, proves the proposition. □

It remains to show the promised lemma:

Lemma 3.3 (Substitution). If $\Gamma, x :: u \vdash e :: t$ and $\Gamma \vdash e' :: u$, then $\Gamma \vdash e[e' \,/\, x] :: t$.

Proof. Once more, we will prove the lemma using an induction on the type derivation for e, inspecting the last derivation step.

If the last step is (e-var), then e is a variable, i.e., $e \equiv y$. If $y \not\equiv x$, then $e[e' \,/\, x] \equiv e$, and the proposition holds. If $y \equiv x$, then $t \equiv u$, because $\Gamma, x :: u \vdash x :: u$. Furthermore, $e[e' \,/\, x] \equiv e'$. Since $\Gamma \vdash e' :: u$, we are done.

If the last step is (e-con), then e is a constructor and unaffected by substitution.

In the case of (e-app), we can apply the induction hypothesis to both function and argument.

All other cases are similar to (e-app). We can make sure that bound variables do not interfere with the substituted variable x by renaming. After that, the substitution only affects the subexpressions that occur in the construct, and we can therefore obtain the desired result directly by applying the induction hypothesis to all subexpressions. □

3.6 Recursive let

We introduce **letrec** as a derived syntactic construct, defined by its translation to a combination of **fix**, **let**, and **case**, as in Figure 3.13. We only show rule (tr-letrec) for the **letrec** construct – all other constructs are unaffected by the translation. The translation assumes that we have tuples available in the language, and we use $\mathsf{sel}(m,n)$ (where $1 \leqslant m \leqslant n$) as an abbreviation for the function that selects the m-th component of an n-tuple. It can be defined as

$$\mathsf{sel}(m,n) \equiv \lambda x \rightarrow \textbf{case } x \textbf{ of } \left(\{x_i\}_{,}^{i\in 1..n}\right) \rightarrow x_m \enspace .$$

We call the language that contains **letrec**, but no (non-recursive) **let**, FCR. Its syntax is defined in Figure 3.12.

Expressions		
e	$::= \ldots$	everything except **let** from Figure 3.1
	$\mid\ \textbf{letrec } \{d_i\}_{;}^{i\in 1..n} \textbf{ in } e$	recursive let

Figure 3.12: Syntax of FCR, modifies the core language FC of Figure 3.1

$$[\![e_{\text{FCR}}]\!]^{\text{rec}} \equiv e_{\text{FC}}$$

$$\frac{z \text{ fresh}}{\begin{array}{l}[\![\textbf{letrec } \{x_i = e_i\}_{;}^{i\in 1..n} \textbf{ in } e_0]\!]^{\text{rec}} \equiv \\ \quad \textbf{let } z = \textbf{fix } \left(\lambda z \rightarrow \{\textbf{let } x_i = \mathsf{sel}(i,n)\ z \textbf{ in}\}^{i\in 1..n} \left(\{[\![e_i]\!]^{\text{rec}}\}_{,}^{i\in 1..n}\right)\right) \\ \quad \textbf{in } \textbf{case } z \textbf{ of } \left(\{x_i\}_{,}^{i\in 1..n}\right) \rightarrow [\![e_0]\!]^{\text{rec}}\end{array}} \quad \text{(tr-letrec)}$$

Figure 3.13: Translation of FCR to FC

We define the operational semantics of **letrec** via the translation into the language without **letrec**, given above. The translation touches only recursive lets, leaving everything else unchanged.

We need a new type judgment for recursive let and prove that it coincides with the type of its translation. The rule is given in Figure 3.14: both expressions bound and the body are checked in an extended environment, which contains the types of the bound variables.

Theorem 3.4 (Correctness of letrec typing). If $e \equiv \textbf{letrec } \{x_i = e_i\}_{;}^{i\in 1..n} \textbf{ in } e_0$, and $\mathrm{K};\Gamma \vdash e :: t$ can be assigned using (e-letrec-val), then $\mathrm{K};\Gamma \vdash [\![e]\!]^{\text{rec}} :: t$.

Proof. The proof is by induction on the derivation of the translated expression.

$$\frac{\begin{array}{c}\Gamma' \equiv \Gamma \;\{, x_i :: t_i\}^{i \in 1..n} \\ \{\Gamma' \vdash e_i :: t_i\}^{i \in 1..n} \qquad \Gamma' \vdash e :: t\end{array}}{\Gamma \vdash \textbf{letrec}\ \{x_i = e_i\}_{;}^{i \in 1..n}\ \textbf{in}\ e :: t} \quad \text{(e-letrec-val)}$$

Figure 3.14: Type checking for recursive let, modifies Figure 3.3

The translation for a recursive let statement involves tuples, thus we assume that appropriate entries are present in the environments. If we set $t_0 \equiv t$, then we know from rule (e-letrec-val) that for all e_i, we have $\Gamma' \vdash e_i :: t_i$, where $\Gamma' \equiv \Gamma, \Gamma''$ with

$$\Gamma'' \equiv \{x_i :: t_i\}_{,}^{i \in 1..n} \quad .$$

The induction hypothesis states that

$$\{\Gamma' \vdash [\![e_i]\!]^{\text{rec}} :: t_i\}^{i \in 0..n} \quad .$$

The first goal is to make a statement about the type of the argument to the **fix** operator. The innermost tuple has the type

$$\Gamma' \vdash (\{[\![e_i]\!]^{\text{rec}}\}_{,}^{i \in 1..n}) :: (\{t_i\}_{,}^{i \in 1..n}) \quad .$$

From the definition of $\mathsf{sel}(i,n)$, it is easy to see that

$$\Gamma, z :: (\{t_i\}_{,}^{i \in 1..n}) \vdash \mathsf{sel}(i,n,z) :: t_i \quad .$$

Therefore, the entire nested non-recursive **let** statement is of type

$$\Gamma, z :: (\{t_i\}_{,}^{i \in 1..n}) \vdash \{\textbf{let}\ x_i = \mathsf{sel}(i,n)\ z\ \textbf{in}\}^{i \in 1..n}\ \left(\{[\![e_i]\!]^{\text{rec}}\}_{,}^{i \in 1..n}\right) :: (\{t_i\}_{,}^{i \in 1..n}) \ ,$$

by repeated application of rule (e-let-val). Subsequently, we use (e-lam) followed by (e-fix) to see that

$$\Gamma \vdash \textbf{fix}\ \left(\lambda z \to \{\textbf{let}\ x_i = \mathsf{sel}(i,n)\ z\ \textbf{in}\}^{i \in 1..n}\ \left(\{[\![e_i]\!]^{\text{rec}}\}_{,}^{i \in 1..n}\right)\right) :: (\{t_i\}_{,}^{i \in 1..n}) \quad .$$

In the following, let

$$e' \equiv \textbf{fix}\ \left(\lambda z \to \{\textbf{let}\ x_i = \mathsf{sel}(i,n)\ z\ \textbf{in}\}^{i \in 1..n}\ \left(\{[\![e_i]\!]^{\text{rec}}\}_{,}^{i \in 1..n}\right)\right) \ ,$$

and we will now inspect the whole translation result

$$\textbf{let}\ z = e'\ \textbf{in case}\ z\ \textbf{of}\ (\{x_i\}_{,}^{i \in 1..n}) \to [\![e_0]\!]^{\text{rec}} \quad .$$

The rule (e-let-val) states that since we already know that $\Gamma \vdash e' :: (\{t_i\}_{,}^{i \in 1..n})$, it suffices to show that

$$\Gamma, z :: (\{t_i\}_{,}^{i \in 1..n}) \vdash \textbf{case}\ z\ \textbf{of}\ (\{x_i\}_{,}^{i \in 1..n}) \to [\![e_0]\!]^{\text{rec}} :: t$$

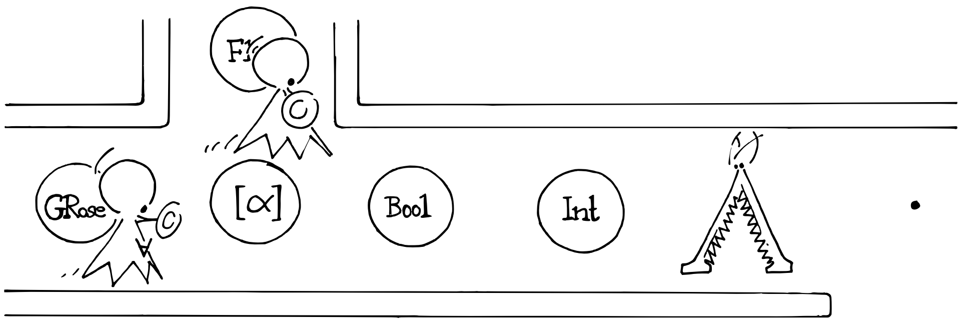
to prove the theorem. The induction hypothesis stated that $\Gamma' \vdash [\![e_0]\!]^{\text{rec}} :: t$, and it does not change anything if we extend the environment Γ' with the binding for z because z does not occur free in e_0 or $[\![e_0]\!]^{\text{rec}}$. It thus remains to apply (e-case) in the right way: because the pattern is the same as above and therefore of type

$$\Gamma, z :: (\{t_i\}_{,}^{i \in 1..n}) \vdash^{pat} (\{x_i\}_{,}^{i \in 1..n}) :: (\{t_i\}_{,}^{i \in 1..n}) \rightsquigarrow \Gamma'' \ ,$$

we are done. □

In Haskell, the keyword **let** is used to denote recursive let. Because FCR contains only recursive let statements, we will do the same from now on and use the **let** keyword instead of **letrec**.

4 Type-indexed Functions



We are now going to leave the known territory of the functional core language. Over the next chapters, we will introduce the theory necessary to compile generic functions in several small steps.

As we have learned in the introduction, generic functions are type-indexed functions. We will therefore, in this chapter, explain type-indexed functions, which are functions that take an explicit type argument and can have behaviour that depends on the type argument. The next chapter extends the class of type arguments we admit in both definition and call sites of type-indexed functions. After that, in Chapter 7, we will really make type-indexed functions generic, such that they work for a significantly larger class of type arguments than those that they are explicitly defined for.

4.1 Exploring type-indexed functions

We call a function **type-indexed** if it takes an explicit type argument and can have behaviour that depends on the type argument. Let us jump immediately to a first example and see what a type-indexed function looks like:

$$\begin{aligned}
&\mathit{add}\ \langle \mathrm{Bool} \rangle &&= (\vee) \\
&\mathit{add}\ \langle \mathrm{Int} \rangle &&= (+) \\
&\mathit{add}\ \langle \mathrm{Char} \rangle &&= \lambda x\ y \to \mathit{chr}\ (\mathit{add}\ \langle \mathrm{Int} \rangle\ (\mathit{ord}\ x)\ (\mathit{ord}\ y)) \ .
\end{aligned}$$

The above function *add* defines an addition function that works for any of the three types Bool, Int, Char. In principle, the definition can be seen as the definition of three independent functions: one function *add* ⟨Bool⟩, which takes two boolean values and returns the logical "or" of its arguments; one function *add* ⟨Int⟩, which performs numerical addition on its two integer arguments; finally, one function *add* ⟨Char⟩, which returns the character of which the numerical code is the sum of the numerical codes of the function's two character arguments. The three functions just happen to have a name consisting of two parts, the first of which is identical for all three of them, and the second of which is the name of a type in special parentheses.

In Generic Haskell, type arguments are always emphasized by special parentheses.

One way in which this type-indexed function differs from three independent functions is its type. The type of *add* is

$$\langle a :: * \rangle \to a \to a \to a \ .$$

Very often, to emphasize that *add* is a type-indexed function, we move the type argument to the left hand side of the type signature:

$$\mathit{add}\ \langle a :: * \rangle :: a \to a \to a \ .$$

This means that *add* has an additional argument, its type argument a, which is of kind $*$ (indeed, Bool, Int, and Char are all three types of kind $*$), and then two arguments of the type indicated by the type argument, yielding a result of the same type. Hence, for a type-indexed function different cases have related types.

According to this type, the function can be called by providing three arguments: one type, and two arguments of that type. Indeed, the calls

add ⟨Bool⟩ *False True*
add ⟨Int⟩ 2 7
add ⟨Char⟩ `'A'` `' '`

would evaluate to *True*, 9, and `'a'` respectively.

What happens if we call *add* on a different type, say Float? For

add ⟨Float⟩ 3.7 2.09 ,

we do not have an appropriate case to select, so all we can do is to fail. But the program fails at compile time! The compiler can check that the function is defined for three types only, and that Float is not among them, and thus report an error.

We could even decide to include the types for which *add* is defined in its type signature and write

$$add\ \langle a :: *, a \in \{\text{Bool}, \text{Int}, \text{Char}\}\rangle :: a \to a \to a\ ,$$

but this would make the type language extremely verbose, and as we will see, it will get verbose enough anyway. Therefore we make the failure to specialize a call to a generic function due to the absence of an appropriate case in the definition another sort of error, which we call **specialization error**. Hence, if we have a Generic Haskell program containing the above definition of *add* and subsequently a call to *add* ⟨Float⟩ like the one above, we get a statically reported specialization error claiming that *add* ⟨Float⟩ cannot be derived, given the cases *add* is defined for.

4.2 Relation to type classes

It should be mentioned once more that what we have seen so far is not very exciting; everything we have done can also – probably even better – be done by using type classes.

Instead of defining *add* as a generic function, we could also define a type class with a method named *add*:

```
class    Add a    where
  add :: a → a → a
instance Add Bool where
  add = (∨)
instance Add Int  where
  add = (+)
instance Add Char where
  add = λx y → chr (ord x + ord y)
```

This code has nearly the same effect as definition of the type-indexed function *add* before. The Haskell type for the class method *add* is

$$add :: (\mathsf{Add}\ a) \Rightarrow a \to a \to a\ ,$$

where the class constraint Add *a* captures the same information as type argument before, namely that there has to be some type *a* of kind $*$ (the kind is implicit in the constraint, because class Add expects an argument of kind $*$). The occurrence of class Add also encodes the fact that the function can only be called on Bool, Int, and Char. If *add* is called on any other type, such as Float, an "instance error" will be reported statically which corresponds directly to the specialization error for the type-indexed functions.

There are two differences. First, the type class can be extended with new instances, whereas the type-indexed function is closed, i.e., it has the cases that are present at the site of its definition, and cannot be extended later. This does not make a huge difference while we consider only single-module programs, but for large programs consisting of multiple modules it can have both advantages and disadvantages. We will discuss the impacts of this design decision as well as the possibilities of "open" type-indexed functions later, in Section 18.4.

Second – and this is a clear advantage of the type classes for now – we do not have to provide the type argument at the call site of the function. We can just write *add* `'A' ' '` and the case for type Char will be selected automatically. In other words, the type argument is inferred for type classes. Type inference for generic functions – in different variations – is the topic of Chapter 13. For now, we assume that all type information is explicitly provided.

4.3 Core language with type-indexed functions FCR+tif

Having introduced the concept of type-indexed functions informally in the context of Haskell, we will now extend the core language of Chapter 3 with type-indexed functions and formally analyze their properties and semantics. This extended language is called FCR+tif, and the syntax modifications with respect to FCR are shown in Figure 4.1.

The language extension introduces a new form of value declarations, for type-indexed functions, using a type-level case statement to pattern match on types. The notation that we have seen in the beginning of this chapter serves as syntactic sugar for the **typecase** construct: the function

$$\begin{aligned}
&add\ \langle \text{Bool} \rangle = (\vee)\\
&add\ \langle \text{Int} \rangle \ \ \ = (+)\\
&add\ \langle \text{Char} \rangle = \lambda x\ y \rightarrow chr\ (ord\ x + ord\ y)
\end{aligned}$$

can be written as

$$\begin{aligned}
&add\ \langle a \rangle = \textbf{typecase}\ a\ \textbf{of}\\
&\qquad\qquad \text{Bool} \rightarrow (\vee)
\end{aligned}$$

Value declarations	
$d \quad ::= x = e$	function declaration, from Figure 3.1
$\quad \mid \ x\ \langle a \rangle = \textbf{typecase}\ a\ \textbf{of}\ \{P_i \to e_i\}_{;}^{i \in 1..n}$	type-indexed function declaration
Expressions	
$e \quad ::= \ldots$	everything from Figures 3.1 and 3.12
$\quad \mid \ x\ \langle A \rangle$	generic application
Type patterns	
$P \quad ::= T$	named type pattern
Type arguments	
$A \quad ::= T$	named type

Figure 4.1: Core language with type-indexed functions FCR+tif, extends language FCR in Figures 3.1 and 3.12

$$\begin{array}{ll} \text{Int} & \to (+) \\ \text{Char} & \to \lambda x\ y \to \mathit{chr}\ (\mathit{ord}\ x + \mathit{ord}\ y) \end{array}$$

in FCR+tif.

Note that the **typecase** construct is tied to the declaration level and, unlike ordinary **case**, may not occur freely in any expression. Furthermore, arms of a case statement distinguish on elaborate patterns, whereas type patterns are for now just names of datatypes (we introduce the syntactic category of type patterns already here because type patterns will be allowed to be more complex later). We assume that a **typecase** is only legal if all type patterns are mutually distinct.

The expression language is extended with generic application that constitutes the possibility to call a type-indexed function (again, only named types are allowed as arguments for now). Type-indexed functions are not first-class in our language: whenever they are called, a type argument must be supplied immediately. Furthermore, type-indexed functions cannot be bound to ordinary variables or passed as arguments to a function. We will explain the reasons for this restriction in Section 11.7. We use the special parentheses to mark type arguments also at the call sites of type-indexed functions.

4.4 Translation and specialization

We will define the semantics of FCR+tif not by providing additional reduction rules for the generic constructs, but by giving a translation into the original core language FCR. We will then prove that the translation is correct by proving that the translation preserves types. The translation is the topic of Figure 4.2.

$$[\![d_{\text{FCR+tif}}]\!]^{\text{tif}}_{\Sigma_1} \equiv \{d_{\text{FCR}}\}^{i\in 1..n}_{;} \rightsquigarrow \Sigma_2$$

$$\frac{\{d_i \equiv \mathsf{cp}(x, T_i) = [\![e_i]\!]^{\text{tif}}\}^{i\in 1..n}}{\begin{array}{l}[\![x\ \langle a\rangle = \textbf{typecase}\ a\ \textbf{of}\ \{T_i \to e_i\}^{i\in 1..n}_{;}]\!]^{\text{tif}}\\ \qquad \equiv \{d_i\}^{i\in 1..n}_{;} \rightsquigarrow \{x\ \langle T_i\rangle\}^{i\in 1..n}_{,}\end{array}} \quad \text{(tr-tif)}$$

$$\frac{}{[\![x = e]\!]^{\text{tif}} \equiv x = [\![e]\!]^{\text{tif}} \rightsquigarrow \varepsilon} \quad \text{(tr-fdecl)}$$

$$[\![e_{\text{FCR+tif}}]\!]^{\text{tif}}_{\Sigma} \equiv e_{\text{FCR}}$$

$$\frac{\begin{array}{c}\Sigma' \equiv \Sigma\ \{,\Sigma_i\}^{i\in 1..n}\\ \{[\![d_i]\!]^{\text{tif}}_{\Sigma'} \equiv \{d_{i,j}\}^{j\in 1..m_i} \rightsquigarrow \Sigma_i\}^{i\in 1..n}\end{array}}{[\![\textbf{let}\ \{d_i\}^{i\in 1..n}_{;}\ \textbf{in}\ e]\!]^{\text{tif}}_{\Sigma} \equiv \textbf{let}\ \{\{d_{i,j}\}^{j\in 1..m_i}_{;}\}^{i\in 1..n}_{;}\ \textbf{in}\ [\![e]\!]^{\text{tif}}_{\Sigma'}} \quad \text{(tr-let)}$$

$$\frac{x\ \langle T\rangle \in \Sigma}{[\![x\ \langle T\rangle]\!]^{\text{tif}}_{\Sigma} \equiv \mathsf{cp}(x, T)} \quad \text{(tr-genapp)}$$

Figure 4.2: Translation of FCR+tif to FCR

We first introduce a rule to translate one FCR+tif declaration into a sequence of FCR declarations. The judgment is of the form

$$[\![d_{\text{FCR+tif}}]\!]^{\text{tif}}_{\Sigma_1} \equiv \{d_{\text{FCR}}\}^{i\in 1..n}_{;} \rightsquigarrow \Sigma_2$$

where the environments Σ_1 and Σ_2 are *signature environments*.

We define the **signature** of a generic function to be the list of types that appear in the patterns of the **typecase** that defines the function. At the moment, type patterns are plainly named types, thus the signature is the list of types for there are cases in the **typecase** construct. For example, the *add* function defined on page 4.1 in Section 4.1 has signature Bool, Int, Char. Note that the *signature* of a type-indexed function is something different than the *type signature* of a (type-indexed) function. The former is a list of named types that is relevant for the translation process, the latter assigns a type to the whole function for type checking purposes.

A **signature environment** contains entries of the form $x\ \langle T\rangle$, indicating that the named type T is in the signature of the type-indexed function x.

In the above-mentioned judgment, the environment Σ_1 is the input environment under which the translation is to take place, Σ_2 is the output environment containing bindings of which we learn during the analysis of the declaration. The environment is only really accessed in the translation of expressions, which takes the form

$$[\![e_{\text{FCR+tif}}]\!]^{\text{tif}}_{\Sigma} \equiv e_{\text{FCR}} \quad .$$

Both translations are understood to be projections – they translate every FCR+tif declaration or expression construct for which there is no special rule and that is also valid in FCR to itself. Therefore, rules that are similar to (tr-fdecl), are left implicit in the translation of expressions.

The declaration of a type-indexed function is translated into a group of declarations with different names. The operation cp is assumed to take a variable and a type name and create a unique variable name out of the two. Here, cp stands for **component**, and we call the translation of a single case of a type-indexed function component henceforth. Furthermore, the signature of the type indexed function defined is stored in the output environment. For a normal function declaration, the expression is translated, and the empty environment is returned.

In a let construct, one or more type-indexed functions can be defined. The environment Σ', containing all the available components, is recursively made visible for the translation of the declarations (i.e., the right hand sides of the declarations may contain recursive calls to the just-defined functions) and to the body of the let statement.

We call the process of translating a call to a type-indexed function **specialization**, because we replace the call to a function which depends on a type argument by a call to a specialized version of the function, for a specific type. The process of specialization thus amounts to selecting the right case of the type-indexed function, or, in other words, performing the pattern match at the type level. In rule (tr-genapp), a generic application is specialized by verifying that the type argument occurs in the signature of the function, and then using the appropriate component of the function as translation.

If the type argument is not an element of the signature of the generic function called, then the translation will fail with a specialization error. In Chapter 5, we will extend the language further to allow a larger class of generic applications to succeed.

4.5 Type checking

To make FCR+tif a "real" language extension, we extend the type rules for the functional core language FCR to cover the constructs for defining and calling type-indexed functions as well. The additional rules are shown in Figure 4.3 and 4.4. Together with the type checking rules of Figure 3.3 and 3.14, they form the type checking judgments for FCR+tif. The judgments still have the same shape, namely

$$\mathrm{K};\Gamma \vdash e :: t\ ,$$

but the environment Γ can now, next to the usual entries of the form $x :: t$, also contain entries of the form $x\ \langle a :: *\rangle :: t$, associating a type with a type-indexed function. As type-indexed and ordinary functions share the same name space, there can only be one active binding for any name x, either as a type-indexed or as an ordinary function.

$$\mathrm{K};\Gamma \vdash e :: t$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash T :: * \\ x\ \langle a :: *\rangle :: t_0 \in \Gamma\end{array}}{\mathrm{K};\Gamma \vdash x\ \langle T\rangle :: t_0[T\ /\ a]}\quad\text{(e-genapp)}$$

$$\frac{\begin{array}{c}\Gamma' \equiv \Gamma\ \{,\Gamma_i\}^{i\in 1..n} \\ \{\Gamma' \vdash^{decl} d_i \rightsquigarrow \Gamma_i\}^{i\in 1..n} \\ \Gamma' \vdash e :: t\end{array}}{\Gamma \vdash \mathbf{let}\ \{d_i\}_{;}^{i\in 1..n}\ \mathbf{in}\ e :: t}\quad\text{(e-let)}$$

Figure 4.3: Type checking for FCR+tif, extends Figure 3.3

The rule for generic application checks that the kind of the type argument is $*$ and that x is a type-indexed function in scope. The type is the type of the type-

$$\mathrm{K};\Gamma_1 \vdash^{decl} d \rightsquigarrow \Gamma_2$$

$$\frac{\vdash e :: t}{\vdash^{decl} x = e \rightsquigarrow x :: t} \quad \text{(d-val)}$$

$$\frac{\{\mathrm{K} \vdash T_i :: *\}^{i \in 1..n} \qquad \mathrm{K}, a :: * \vdash t :: * \qquad \mathrm{K};\Gamma \vdash e_i :: t[T_i \mathbin{/} a]}{\mathrm{K};\Gamma \vdash^{decl} x\ \langle a \rangle = \textbf{typecase}\ a\ \textbf{of}\ \{T_i \to e_i\}_{;}^{i \in 1..n} \rightsquigarrow x\ \langle a :: * \rangle :: t} \quad \text{(d-tif)}$$

Figure 4.4: Type checking for declarations in FCR+tif, extends Figure 4.3

indexed function, with the formal type argument in its type substituted by the actual type argument of the application.

The former rule (e-letrec-val) is now obsolete and replaced by the more general (e-let) that allows for both value and type-indexed function declarations. It makes use of a subsidiary judgment for declarations of the form

$$\mathrm{K};\Gamma_1 \vdash^{decl} d \rightsquigarrow \Gamma_2$$

that checks a declaration under environments K and Γ_1 and results in an environment Γ_2 containing possible new bindings introduced by the declaration. The two rules for that judgment are presented in Figure 4.4.

The rule (d-val) is for value declarations. It is easy to see that the new rules for let-statements are generalizations that coincide with the old rules for the case that all declarations are of the form $x = e$.

The second rule, (d-tif), is for declarations of type-indexed functions. All type patterns must be of kind $*$. There must be a type t of kind $*$, containing a type variable a of kind $*$, and all expressions e_i in the arms of the typecase must have an instance of this type t, with a substituted by the type pattern T_i. The type t that has this characteristic is then returned as the type of the type-indexed function in the resulting environment.

We now want to show that the translation is correct, i.e., that it preserves type correctness, or in this case even the exact types of expressions. However, before we can proceed to the theorem, we first have to extend the translation to environments. Signature environments do not exist in FCR, and type environments cannot contain type signatures for type-indexed functions. Translations are introduced in Figure 4.5, and judgments are of the forms

$$[\![\Gamma_{\text{FCR+tif}}]\!]^{\text{tif}} \equiv \Gamma_{\text{FCR}}$$

$$\frac{}{[\![\varepsilon]\!]^{\text{tif}} \equiv \varepsilon} \quad \text{(tif-gam-1)} \qquad \frac{}{[\![\Gamma, x :: t]\!]^{\text{tif}} \equiv [\![\Gamma]\!]^{\text{tif}}, x :: t} \quad \text{(tif-gam-2)}$$

$$\frac{}{[\![\Gamma, x \ \langle a :: *\rangle :: t]\!]^{\text{tif}} \equiv [\![\Gamma]\!]^{\text{tif}}} \quad \text{(tif-gam-3)}$$

$$[\![\Sigma_{\text{FCR+tif}}]\!]^{\text{tif}}_{\Gamma_{\text{FCR+tif}}} \equiv \Gamma_{\text{FCR}}$$

$$\frac{}{[\![\varepsilon]\!]^{\text{tif}} \equiv \varepsilon} \quad \text{(tif-sig-1)}$$

$$\frac{x \ \langle a :: *\rangle :: t \in \Gamma}{[\![\Sigma, x \ \langle T\rangle]\!]^{\text{tif}}_{\Gamma} \equiv [\![\Sigma]\!]^{\text{tif}}_{\Gamma}, \mathsf{cp}(x, T) :: t[T \ / \ a]} \quad \text{(tif-sig-2)}$$

Figure 4.5: Translation of FCR+tif environments to FCR type environments

$$\begin{aligned} \llbracket \Gamma_{\textsc{fcr}+\text{tif}} \rrbracket^{\text{tif}} &\equiv \Gamma_{\textsc{fcr}} \\ \llbracket \Sigma_{\textsc{fcr}+\text{tif}} \rrbracket^{\text{tif}}_{\Gamma_{\textsc{fcr}+\text{tif}}} &\equiv \Gamma_{\textsc{fcr}} \ . \end{aligned}$$

The former filters type signatures of type-indexed functions from a type environment, whereas the latter translates a signature environment into type signatures for the associated components: every type that is in the signature of a type-indexed function is translated into a type assumption for the corresponding component of the function. We use the abbreviation

$$\llbracket \Gamma; \Sigma \rrbracket^{\text{tif}}_{\Gamma} \equiv \llbracket \Gamma \rrbracket^{\text{tif}}, \llbracket \Sigma \rrbracket^{\text{tif}}_{\Gamma} \ .$$

Note that the translation of the signature environment produces bindings for components only, and they never clash with the bindings from the translated type environment.

Now we have everything we need for the theorem:

Theorem 4.1 (Correctness of **FCR+tif).** If e is a FCR+tif expression with $K; \Gamma \vdash e :: t$, then $K; \llbracket \Gamma; \Sigma \rrbracket^{\text{tif}}_{\Gamma} \vdash \llbracket e \rrbracket^{\text{tif}}_{\Sigma} :: t$, assuming that Σ is a signature environment such that $\llbracket e \rrbracket^{\text{tif}}_{\Sigma}$ exists.

Corollary 4.2. If e is a FCR+tif expression with $K; \Gamma \vdash e :: t$ with no type-indexed bindings in Γ, and $\Sigma \equiv \varepsilon$, then $K; \Gamma \vdash \llbracket e \rrbracket^{\text{tif}}_{\varepsilon} :: t$.

In other words, if the translation of an FCR+tif expression into FCR succeeds, the resulting expression has the same type as the original expression. The corollary, which follows immediately from the theorem, emphasizes the special case where Σ is empty. In this situation exactly the same environments can be used to assign t to both e and its translation.

Proof of the theorem. An induction on the type derivation for e will do the trick once more. Only the new cases that are possible for the last derivation step are interesting: if the last step is (e-genapp), then $e \equiv x \ \langle T \rangle$, and we know by (tr-genapp) that $x \ \langle T \rangle$ is in Σ. Furthermore, (e-genapp) ensures that $x \ \langle a :: * \rangle :: t_0$ is in Γ, where $t \equiv t_0[T \ / \ a]$. Now, $\llbracket e \rrbracket^{\text{tif}}_{\Sigma} \equiv \mathsf{cp}(x, T)$, and $\mathsf{cp}(x, T) :: t_0[T \ / \ a]$ is in $\llbracket \Gamma; \Sigma \rrbracket^{\text{tif}}_{\Gamma}$.

If the last step of the type derivation for e is (e-let), we first need to extend the correctness theorem (and the induction) to the translation of declarations. We will prove the following property: if $\llbracket d \rrbracket^{\text{tif}}_{\Sigma} \equiv \{d_i\}^{i \in 1..n} \rightsquigarrow \Sigma'$ and both $K; \Gamma \vdash^{decl} d \rightsquigarrow \Gamma'$ and $\{K; \llbracket \Gamma; \Sigma \rrbracket^{\text{tif}}_{\Gamma} \vdash^{decl} d_i \rightsquigarrow \Gamma_i\}^{i \in 1..n}$, then $\llbracket \Gamma'; \Sigma' \rrbracket^{\text{tif}}_{\Gamma'} \equiv \{\Gamma_i\}^{i \in 1..n}_{,}$.

If the derivation for d ends in the rule (d-val), then $d \equiv x = e$. In this case, $\Sigma' \equiv \varepsilon$, $n \equiv 1$, and $d_1 \equiv x = \llbracket e \rrbracket^{\text{tif}}_{\Sigma}$. By induction hypothesis, we know that if $\Gamma \vdash e :: t$, then $\llbracket \Gamma; \Sigma \rrbracket^{\text{tif}}_{\Gamma} \vdash \llbracket e \rrbracket^{\text{tif}}_{\Sigma} :: t$. Therefore, the two applications of $\vdash^{decl}$ are

$$K; \Gamma \qquad \vdash^{decl} x = e \qquad \rightsquigarrow x :: t$$

and

$$K; [\![\Gamma; \Sigma]\!]^{\mathrm{tif}}_{\Gamma} \vdash^{decl} x = [\![e]\!]^{\mathrm{tif}}_{\Sigma} \rightsquigarrow x :: t\,,$$

thus $\Gamma' \equiv \Gamma_1 \equiv x :: t$. This implies $[\![\Gamma'; \Sigma]\!]^{\mathrm{tif}}_{\Gamma'} \equiv [\![\Gamma'; \varepsilon]\!]^{\mathrm{tif}}_{\Gamma'} \equiv \Gamma_1$.

If the derivation for d ends in the rule (d-tif), then

$$d \equiv x\ \langle a \rangle = \textbf{typecase}\ a\ \textbf{of}\ \{T_i \to e_i\}^{i \in 1..n}_{;}\ .$$

Here, we apply the induction hypothesis to the e_i, getting the e_i and their translations $[\![e_i]\!]^{\mathrm{tif}}_{\Sigma}$ have the same type, say $\Gamma \vdash e_i :: t_i$ and $[\![\Gamma; \Sigma]\!]^{\mathrm{tif}}_{\Gamma} \vdash [\![e_i]\!]^{\mathrm{tif}}_{\Sigma} :: t_i$. We know from (d-tif) that there is a t such that $t_i \equiv t[T_i\ /\ a]$. Furthermore, in this situation $d_i \equiv \mathsf{cp}(x, T_i) = [\![e_i]\!]^{\mathrm{tif}}_{\Sigma}$, and $\Sigma' \equiv \{x\ \langle T_i \rangle\}^{i \in 1..n}_{,}$. Next to that, we can conclude that $\Gamma' \equiv x\ \langle a :: * \rangle :: t$ and $\Gamma_i \equiv \mathsf{cp}(x, T_i) :: t_i$. We thus have to show that

$$[\![x\ \langle a :: * \rangle :: t; \{x\ \langle T_i \rangle\}^{i \in 1..n}_{,}]\!]^{\mathrm{tif}}_{x\ \langle a :: * \rangle :: t} \equiv \{\mathsf{cp}(x, T_i) :: t_i\}^{i \in 1..n}_{,}\,,$$

but this follows immediately from the rules in Figure 4.5 for the translation of signature environments.

We can now cover the case of the proof of the theorem where the derivation for e concludes on an application of (e-let).

Here, we know that $e \equiv \textbf{let}\ \{d_i\}^{i \in 1..n}_{;}\ \textbf{in}\ e_0$, and

$$[\![e]\!]^{\mathrm{tif}}_{\Sigma} \equiv \textbf{let}\ \{\{d_{i,j}\}^{j \in 1..m_i}_{;}\}^{i \in 1..n}_{;}\ \textbf{in}\ e'_0$$

where $[\![e_0]\!]^{\mathrm{tif}}_{\Sigma'} \equiv e'_0$, and $\Sigma' \equiv \Sigma\ \{, \Sigma_i\}^{i \in 1..n}$ with

$$\{[\![d_i]\!]^{\mathrm{tif}}_{\Sigma'} \equiv \{d_{i,j}\}^{j \in 1..m_i} \rightsquigarrow \Sigma_i\}^{i \in 1..n}\ .$$

We now apply the induction hypothesis to e_0 and the d_i. For e_0, we get that both $\Gamma' \vdash e_0 :: t$ and $[\![\Gamma'; \Sigma']\!]^{\mathrm{tif}}_{\Gamma'} \vdash e'_0 :: t$, where $\Gamma' \equiv \Gamma\ \{, \Gamma_i\}^{i \in 1..n}$ and $\Gamma' \vdash^{decl} d_i \rightsquigarrow \Gamma_i$).

For the declarations, we use the correctness property that we have proved above, which yields

$$\{[\![\Gamma_i; \Sigma_i]\!]^{\mathrm{tif}}_{\Gamma_i} \equiv \{\Gamma_{i,j}\}^{j \in 1..m_i}_{,}\}^{i \in 1..n}\,,$$

where $[\![\Gamma'; \Sigma']\!]^{\mathrm{tif}}_{\Gamma'} \vdash^{decl} d_{i,j} \rightsquigarrow \Gamma_{i,j}$. Observing that

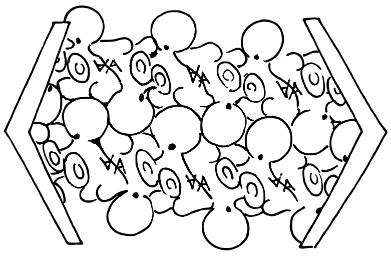
$$[\![\Gamma'; \Sigma']\!]^{\mathrm{tif}}_{\Gamma'} \equiv [\![\Gamma; \Sigma]\!]^{\mathrm{tif}}_{\Gamma}\ \{, [\![\Gamma_i; \Sigma_i]\!]^{\mathrm{tif}}_{\Gamma_i}\,, \}^{i \in 1..n}$$

an application of rule (e-letrec-val) results in

$$[\![\Gamma; \Sigma]\!]^{\mathrm{tif}}_{\Gamma} \vdash \textbf{let}\ \{\{d_{i,j}\}^{j \in 1..m_i}_{;}\}^{i \in 1..n}_{;}\ \textbf{in}\ e'_0 :: t\,,$$

which is precisely what we need. □

5 Parametrized Type Patterns



5.1 Goals

In the type-indexed functions that we have treated so far, one deficiency stands out: types of a kind other than $*$ are nowhere allowed, not in type patterns nor in type arguments. A consequence is that also composite types of kind $*$ – such as $[\mathrm{Int}]$, where the list type constructor $[\,]$, of kind $* \to *$, is involved – are not allowed.

What if we want a type-indexed function that computes something based on a data structure of kind $* \to *$, for example the size of a data structure, i.e., the number of "elements" in that structure?

A definition such as

$$\begin{array}{lll}
\mathit{size}\ \langle[\alpha]\rangle & x & = \mathit{length}\ x \\
\mathit{size}\ \langle\mathrm{Maybe}\ \alpha\rangle & \mathit{Nothing} & = 0 \\
\mathit{size}\ \langle\mathrm{Maybe}\ \alpha\rangle & (\mathit{Just}\ \_) & = 1 \\
\mathit{size}\ \langle\mathrm{Tree}\ \alpha\rangle & \mathit{Leaf} & = 0 \\
\mathit{size}\ \langle\mathrm{Tree}\ \alpha\rangle & (\mathit{Node}\ \ell\ x\ r) & = \mathit{size}\ \langle\mathrm{Tree}\ \alpha\rangle\ \ell + 1 + \mathit{size}\ \langle\mathrm{Tree}\ \alpha\rangle\ r
\end{array}$$

might do, if we assume that Tree is a Haskell datatype defined as

$$\textbf{data}\ \mathrm{Tree}\ (a :: *) = \mathit{Leaf} \mid \mathit{Node}\ (\mathrm{Tree}\ a)\ a\ (\mathrm{Tree}\ a)\ \ .$$

For the definition, we assume that patterns of the form $T\ \alpha$ are allowed, where T is a named type of kind $* \rightarrow *$, and α is a variable. All patterns that occur in the definition are of this form.

But that is still too limiting: a type-indexed function that works on types of kind $*$ may as well work on some data structures of higher kind. Recall our example from last chapter, the *add* function:

$$\begin{array}{llll}
\mathit{add}\ \langle \mathrm{Bool}\rangle & & & = (\vee) \\
\mathit{add}\ \langle \mathrm{Int}\rangle & & & = (+) \\
\mathit{add}\ \langle \mathrm{Char}\rangle & x & y & = \mathit{chr}\ (\mathit{ord}\ x + \mathit{ord}\ y)\ \ .
\end{array}$$

Given that we know how to add two values of some type *t*, we can also add two values of type Maybe *t*, by treating *Nothing* as exceptional value:

$$\begin{array}{llll}
\mathit{add}\ \langle \mathrm{Maybe}\ \alpha\rangle & \mathit{Nothing} & \_ & = \mathit{Nothing} \\
\mathit{add}\ \langle \mathrm{Maybe}\ \alpha\rangle & \_ & \mathit{Nothing} & = \mathit{Nothing} \\
\mathit{add}\ \langle \mathrm{Maybe}\ \alpha\rangle & (\mathit{Just}\ x) & (\mathit{Just}\ y) & = \mathit{Just}\ (\mathit{add}\ \langle \alpha\rangle\ x\ y)\ ,
\end{array}$$

The knowledge of how to add two values of the argument type of Maybe is hidden in the reference to *add* $\langle\alpha\rangle$ in the final case, and we still have to make sure that we get access to that information somehow.

We could also extend the function *add* to lists as pointwise addition:

$$\begin{array}{llll}
\mathit{add}\ \langle [\alpha]\rangle & x & y & \\
\quad \mid \mathit{length}\ x == \mathit{length}\ y & & & = \mathit{map}\ (\mathit{uncurry}\ (\mathit{add}\ \langle\alpha\rangle))\ (\mathit{zip}\ x\ y) \\
\quad \mid \mathit{otherwise} & & & = \mathit{error}\ \texttt{"args must have same length"}\ \ .
\end{array}$$

We do not even have to restrict ourselves to kind $*$ and $* \rightarrow *$ datatypes. We can do pointwise addition for pairs, too:

$$\mathit{add}\ \langle(\alpha,\beta)\rangle \quad (x_1,x_2)\ \ (y_1,y_2)\ = (\mathit{add}\ \langle\alpha\rangle\ x_1\ y_1, \mathit{add}\ \langle\beta\rangle\ x_2\ y_2)\ \ .$$

Now our function *add* has arms which involve type constructors of kinds $*$, $* \rightarrow *$, and $* \rightarrow * \rightarrow *$, all at the same time.

Perhaps surprisingly, we need not just one, but three significant extensions to the simple type-indexed functions introduced in Chapter 4, to be able to successfully handle the above examples.

The first requirement is clear: type patterns must become more general. Instead of just named types of kind $*$, we will admit named types of arbitrary kind, applied to type variables in such a way that the resulting type is of kind $*$ again. This addition will be described in more detail in Section 5.2.

Secondly, and this is the most difficult part, we need to introduce the notion of *dependencies* between type-indexed functions. A dependency arises if in the definition of one type-indexed function, another type-indexed function (including the function itself) is called, with a *variable* type as type argument. Dependencies must be tracked by the type system, so the type system must be extended accordingly. All this is the topic of Section 5.3.

Last, to be of any use, we must also extend the specialization mechanism. Until now, we could only handle *calls* to type-indexed functions for named types of kind $*$. Now, we want to be able to call *size* $\langle[\mathrm{Int}]\rangle$ or *add* $\langle(\mathrm{Char},[\mathrm{Bool}])\rangle$. Thus, type arguments have to be generalized so that they may contain type applications as well. This is explained in Section 5.4.

In the rest of this chapter, we will describe all of these extensions in detail by means of examples. In the next chapter, we will formally extend our core language with the constructs necessary to support the extensions.

5.2 Parametrized type patterns

We now allow patterns such as $\langle[\alpha]\rangle$ or $\langle\mathrm{Either}\ \alpha\ \beta\rangle$ in the definitions of type-indexed functions. A pattern must be of kind $*$, and of the form that a named type constructor is applied to variables to satisfy the type constructor. Thus $\langle\mathrm{Either}\ \alpha\rangle$ is not allowed because Either is only partially applied. Of course, types of kind $*$ are a special case of this rule: the type pattern $\langle\mathrm{Int}\rangle$ is the nullary type constructor Int applied to no type variables at all.

All type variables in type patterns have to be distinct, just as we require variables in ordinary patterns to be distinct. A pattern such as $\langle(\alpha,\alpha)\rangle$ is illegal. Also, we do *not* allow **nested** type patterns: $\langle[\mathrm{Int}]\rangle$ is forbidden, and so is $\langle\mathrm{Either}\ \alpha\ \mathrm{Char}\rangle$. The top-level type constructor is the only named type occurring in a type pattern, the rest are all type variables. This restriction on type patterns is not essential. One could allow nested type patterns, or multiple patterns for the same type, such as they are allowed in Haskell **case** statements. This would significantly complicate the future algorithms for the translation of type-indexed functions with issues that are not directly related to generic programming. With our restriction, performing pattern matching on the cases of a type-indexed definition remains as simple as possible: comparing the top-level constructors is sufficient to find and select the correct branch.

We retain the notions of *signature*, *specialization*, and *component*, all defined in Section 4.4. The signature of a type-indexed function is the set of named types occurring in the type patterns. For *size*, the signature is $[\,]$, Maybe, Tree. For *add*, including the new cases, the signature consists of the six types Bool, Int, Char, Maybe, $[\,]$, $(,)$. Specialization is the process of translating a call of a generic function. A component is the result of translating a single arm of a **typecase** construct.

This translation to components can be very simple in some cases. For instance, the arm of the *size* function for lists,

$$size\ \langle[\alpha]\rangle\ x = length\ x\ ,$$

can be translated into a component $\mathtt{cp}(size, [\,])\ x = length\ x$ almost in the same way as we translate arms for types of kind $*$. Note that components are always generated for a type-indexed function and a named type. In this case, the function is *size*, and the named type $[\,]$. As the signature of a type-indexed function can contain types of different kind (compare with the example signature for *add* above), also components can be created for named types of different types. The variables in the type patterns do not occur in the translation.

In this case, the translation is easy because the variable α is not used on the right hand side. Things get more difficult if the variables in the type patterns are used in generic applications on the right hand sides, and we will discuss that in the following section.

5.3 Dependencies between type-indexed functions

When a type-indexed function is called within the definition of another type-indexed function, we must distinguish different sorts of calls, based on the type argument: calls with type arguments that are constant are treated differently from calls where the type argument contains variables.

Let us look at the *add* function once more, in particular at the Int and Char cases of the function:

$$\begin{aligned} &add\ \langle \text{Int}\rangle && = (+) \\ &add\ \langle \text{Char}\rangle\ x\ y && = chr\ (ord\ x + ord\ y) \end{aligned}$$

The *ord* of a character is an integer, therefore we could equivalently have written

$$add\ \langle \text{Char}\rangle\ x\ y \qquad = chr\ \big(add\ \langle \text{Int}\rangle\ (ord\ x)\ (ord\ y)\big)$$

During translation, we can generate a component $\mathtt{cp}(add, \text{Char})$ as usual: we specialize the call *add* $\langle$Int$\rangle$ on the right hand side to refer to $\mathtt{cp}(add, \text{Int})$. The type argument Int is statically known during the translation of the function definition, therefore the compiler can locate and access the appropriate component.

On the other hand, in the case for lists

$$
\begin{aligned}
&\mathit{add}\ \langle[\alpha]\rangle\quad x\ y\\
&\quad \mid \mathit{length}\ x \mathrel{==} \mathit{length}\ y = \mathit{map}\ \big(\mathit{uncurry}\ (\mathit{add}\ \langle\alpha\rangle)\big)\ (\mathit{zip}\ x\ y)\\
&\quad \mid \mathit{otherwise} \qquad\qquad\ \ = \mathit{error}\ \texttt{"args must have same length"}\ ,
\end{aligned}
$$

we *cannot* simply generate a component $\mathtt{cp}(\mathit{add}, [\,])$, because we have to specialize the call $\mathit{add}\ \langle\alpha\rangle$, without knowing at the definition site of *add* what α will be. This information is only available where the function *add* is called. Nevertheless, it is desirable that we can translate the definition of a type-indexed function without having to analyze where and how the function is called.

The solution to this problem is surprisingly simple: we say that the function *add* is a **dependency** of *add*. A dependency makes explicit that information is missing. This information is needed during the translation. In the result of the translation, this information is provided in the form of an additional function argument passed to the components of *add*. The right hand sides can then access this argument.

To be more precise: a component for a type argument involving free variables expects additional arguments – one for each combination of variable and function among the dependencies. In our example, there is one variable α, and one dependency, *add*, so there is one additional argument, which we will succinctly call $\mathtt{cp}(\mathit{add}, \alpha)$, because it tells us how to add values of type α and abstracts from an unknown component. The component that is generated will be

$$
\begin{aligned}
&\mathtt{cp}(\mathit{add}, [\,])\ \mathtt{cp}(\mathit{add}, \alpha)\ x\ y\\
&\quad \mid \mathit{length}\ x \mathrel{==} \mathit{length}\ y = \mathit{map}\ (\mathit{uncurry}\ \mathtt{cp}(\mathit{add}, \alpha))\ (\mathit{zip}\ x\ y)\\
&\quad \mid \mathit{otherwise} \qquad\qquad\ \ = \mathit{error}\ \texttt{"args must have same length"}\ .
\end{aligned}
$$

In this example, the type-indexed function *add* depends on itself – we say that it is *reflexive*. This reflexivity occurs frequently with type-indexed functions, because it corresponds to the common case that a function is defined using direct recursion. Still, type-indexed function can depend on arbitrary other type-indexed functions. These can, but do not have to include the function itself.

Dependencies of type-indexed functions are reflected in their *type signatures*. Previously, *add* had the type

$$\mathit{add}\ \langle a :: *\rangle :: a \to a \to a\ .$$

This type is no longer adequate – we use a new syntax,

$$\mathit{add}\ \langle a :: *\rangle :: (\mathit{add}) \Rightarrow a \to a \to a\ .$$

In addition to the old type, we store the dependency of *add* on itself in the type signature. This type signature is a formal way to encode all type information about *add* that is necessary. In general, the **type signature** of a type-indexed function consists of the name of the function and its type argument to the left of the double colon ::, and a list of function names that constitute **dependencies** of the function, followed by a double arrow ⇒ and the function's **base type** to the right of the double colon. Sometimes, we use the term *type signature* to refer only to the part to the right of the double colon.

According to the definitions above, the function *add* has one dependency: the function *add* itself. The base type of *add* is $a \to a \to a$.

This generalized form of type signature with a list of dependencies still does not enable us to cover the types of all generic functions we would like to write. We will extend upon the material of this chapter later, in Chapter 9.

There is an algorithm that allows us – given the type signature – to determine the type of $add\ \langle A\rangle$ for any type argument (or pattern) A. This algorithm, called gapp, will be discussed in detail in Section 6.3.

Now, let us look at some example types for specific applications of *add*: for a constant type argument such as Int, or [Char], the dependencies are ignored, and the types are simply

$$\begin{aligned} &add\ \langle \mathrm{Int}\rangle &&:: \mathrm{Int} &&\to \mathrm{Int} &&\to \mathrm{Int}\\ &add\ \langle [\mathrm{Char}]\rangle &&:: [\mathrm{Char}] &&\to [\mathrm{Char}] &&\to [\mathrm{Char}]\ . \end{aligned}$$

These types are for specific instances of the type-indexed functions, and they can be derived automatically from above type signature. In general, for any type argument A that is dependency-variable free, we have

$$add\ \langle A\rangle :: A \to A \to A\ .$$

Dependencies have an impact on the type of a generic application once variables occur in the type argument. For instance, if the type argument is $[\alpha]$, the resulting type is

$$add\ \langle [\alpha]\rangle :: \forall a :: *.\ (add\ \langle\alpha\rangle :: a \to a \to a) \Rightarrow [a] \to [a] \to [a]\ .$$

We call the part in the parentheses to the left of the double arrow a **dependency constraint**. In this case, $add\ \langle\alpha\rangle :: a \to a \to a$ is a dependency constraint. This means that we can assign the type $[a] \to [a] \to [a]$ to the call, but only under the condition that we know how $add\ \langle\alpha\rangle$ is defined, and that $add\ \langle\alpha\rangle$ has to be of type $a \to a \to a$, which is the base type of *add*. Again, this is the type for the generic application $add\ \langle[\alpha]\rangle$, and the type can be derived from the type signature for *add* given above using the gapp algorithm from Section 6.3.

Dependency constraints are comparable to type class constraints (Wadler and Blott 1989; Jones 1994) in Haskell, or perhaps even better to constraints for implicit parameters (Lewis *et al.* 2000). A dependency constraint encodes an implicit argument that must be provided. We will see later that in the translation to the core language FCR, these implicit arguments are turned into explicit function arguments.

Recall the definition of *add* for $[\alpha]$:

```
add ⟨[α]⟩ x y
    | length x == length y = map (uncurry (add ⟨α⟩)) (zip x y)
    | otherwise            = error "args must have same length" .
```

On the right hand side, there is a call to *add* $\langle\alpha\rangle$. This occurrence of *add* $\langle\alpha\rangle$ has the type

$$\forall a :: *.\ (add\ \langle\alpha\rangle :: a \to a \to a) \Rightarrow a \to a \to a\ .$$

The call has the type $a \to a \to a$, but at the same time *introduces* a dependency on *add* $\langle\alpha\rangle$ of type $a \to a \to a$. This reflects the fact that the generic application *add* $\langle\alpha\rangle$ does only make sense in a context where the lacking information is provided somehow. The translation refers to $\mathtt{cp}(add,\alpha)$, which must be in scope.

The whole right hand side of the definition – because it is the arm of *add* for $[\alpha]$ – must have the aforementioned type for *add* $\langle[\alpha]\rangle$, which is

$$\forall a :: *.\ (add\ \langle\alpha\rangle :: a \to a \to a) \Rightarrow [a] \to [a] \to [a]\ .$$

The right hand side thus *may* depend on *add* $\langle\alpha\rangle$. The type pattern of the arm *eliminates* the dependency. The translation will provide the function argument $\mathtt{cp}(add,\alpha)$, which is then in scope for the right hand side of the component.

In general, we say that dependency constraints are **introduced** by a call to a generic function on a type argument involving variables, and they are **eliminated** or **satisfied** by a type pattern in the definition of a type-indexed function.

For now, type patterns are the only way to eliminate dependency constraints. This implies that there is not much sense in calling type-indexed functions on type arguments with variables except while defining a type-indexed function. We will learn about another mechanism to eliminate dependency constraints in Chapter 8.

Figure 5.1 summarizes example types for generic applications of the *add* function, for different sorts of type arguments. In all cases, the metavariable A is supposed to be free of variables, but we assume that it is of different kind in each case: $*$, $* \to *$, $* \to * \to *$, and finally $(* \to *) \to *$, applied to variables of suitable kind to make the type argument as a whole a kind $*$ type. For each variable in the type pattern, there is one dependency constraint. We call the variables in the pattern (denoted by Greek letters) **dependency variables**. For each

$$\begin{array}{llll}
\mathit{add}\ \langle A :: *\rangle :: & & A & \to A \quad \to A \\
\mathit{add}\ \langle A\ (\alpha :: *) :: *\rangle & & & \\
\quad :: \forall a :: *. & (\mathit{add}\ \langle\alpha\rangle :: a \to a \to a) \Rightarrow & A\ a & \to A\ a \quad \to A\ a \\
\mathit{add}\ \langle A\ (\alpha :: *)\ (\beta :: *) :: *\rangle & & & \\
\quad :: \forall (a :: *)\ (b :: *). & (\mathit{add}\ \langle\alpha\rangle :: a \to a \to a, & & \\
 & \ \mathit{add}\ \langle\beta\rangle :: b \to b \to b) \Rightarrow & A\ a\ b & \to A\ a\ b \to A\ a\ b \\
\mathit{add}\ \langle A\ (\alpha :: * \to *) :: *\rangle & & & \\
\quad :: \forall a :: * \to *. & (\mathit{add}\ \langle\alpha\ (\gamma :: *)\rangle :: & & \\
 & \quad \forall c :: *. (\mathit{add}\ \langle\gamma\rangle :: c \to c \to c) \Rightarrow a\ c \to a\ c \to a\ c) & & \\
 & \quad \Rightarrow & A\ a & \to A\ a \to A\ a \ .
\end{array}$$

Figure 5.1: Types for generic applications of *add* to type arguments of different form

dependency variable, one type variable of the same kind is introduced. In the examples, α is associated with a, and β with b, and γ with c. It may seem strange that we do not just use the same variables, but rather distinguish the quantified type variables from the dependency variables (we distinguish the two sorts of variables even on a syntactic level, which is why we use Greek letters to denote dependency variables). The reason is that in the more general situation that will be discussed in Chapter 9, we will allow multiple type variables to be associated with one dependency variable. Dependency variables are only allowed in type patterns of type-indexed function definitions and in type arguments in dependency constraints and calls to type-indexed functions – everywhere between the special type parentheses $\langle\cdot\rangle$.

If a dependency variable is of higher kind – as can be seen in the last example for α of kind $* \to *$ – the associated dependency constraint is nested: it introduces local dependency variables – such as γ in the example – and the type of the dependency is itself a dependency type. How can this nested dependency type be understood? The call $\mathit{add}\ \langle A\ (\alpha :: * \to *)\rangle$ is of type $A\ a \to A\ a \to A\ a$, but depends on an implicit argument called $\mathit{add}\ \langle\alpha\ \gamma\rangle$, which is of type

$$\forall c :: *. (\mathit{add}\ \langle\gamma\rangle :: c \to c \to c) \Rightarrow a\ c \to a\ c \to a\ c\ .$$

This $\mathit{add}\ \langle\alpha\ \gamma\rangle$ may thus itself depend on some function $\mathit{add}\ \langle\gamma\rangle$ of some type $c \to c \to c$, and must, given this function, be of type $a\ c \to a\ c \to a\ c$. Note that the form of the type that the dependency has, is itself very similar to the type of the second example, where A is of kind $* \to *$.

In theory, there is no limit to the kinds that may occur in the type arguments. However, more complex kinds rarely occur in practice.

The function *add* represents the most common case of dependency: *add* depends on itself, and on nothing else. This corresponds to a normal function which is defined by means of recursion on itself. But a type-indexed function need not depend on itself, or it may depend on other type-indexed functions. We have already seen a function of the first category, namely *size* (defined on page 53). Its type signature is

$$size \ \langle a :: * \rangle :: () \Rightarrow a \to \mathrm{Int} \ .$$

We often omit the empty list of dependencies and write

$$size \ \langle a :: * \rangle :: a \to \mathrm{Int} \,,$$

which coincides with the old form of type signatures, that we used when we did not know about dependencies.

Note that although *size* calls itself recursively in the Tree arm,

$$size \ \langle \mathrm{Tree} \ \alpha \rangle \ (Node \ \ell \ x \ r) = size \ \langle \mathrm{Tree} \ \alpha \rangle \ \ell + 1 + size \ \langle \mathrm{Tree} \ \alpha \rangle \ r \,,$$

it does not depend on itself. The position of a dependency variable in a type argument determines whether or not (and if yes, which) dependency constraints are needed: if the type argument is of the form $A_0 \ \{A_i\}^{i \in 1..n}$, and A_0 does not contain any further application, then A_0 is called the **head**. We also write $\mathsf{head}(A)$ to denote the head of a type argument. In *size* $\langle \mathrm{Tree} \ \alpha \rangle$, the head of the type argument is Tree. In *add* $\langle \alpha \rangle$, the head of the type argument is α.

If a dependency variable α is the head of A, then a call $x \ \langle A \rangle$ introduces a dependency constraint on $x \ \langle \alpha \rangle$. If α occurs in A, but not as the head of A, then the call x introduces dependency constraints according to the dependencies of x: for each function y_k that is a dependency of x, a dependency constraint on $y_k \ \langle \alpha \rangle$ is introduced.

Let us map this abstract rule to our examples. The definition of *add* (see page 54) contains, for instance, a call to *add* $\langle \alpha \rangle$ on the right hand side of the case for *add* $\langle [\alpha] \rangle$. The dependency variable α is in the head position in this call, therefore a dependency constraint on *add* $\langle \alpha \rangle$ is introduced (amounting to a reference to $\mathsf{cp}(add, \alpha)$ in the translation). As a result, *add* must depend on itself, because only then the dependency constraint can be satisfied. The translation will then provide the additional function argument $\mathsf{cp}(add, \alpha)$ for the component $\mathsf{cp}(add, [\,])$ resulting from case *add* $\langle [\alpha] \rangle$.

On the other hand, in the situation of *size*, the head of the type argument in the call *size* $\langle \mathrm{Tree} \ \alpha \rangle$ is Tree, and α occurs somewhere else in the type argument. Therefore, this application of *size* does not necessarily force a dependency for *size* $\langle \alpha \rangle$. Instead, the call introduces dependency constraints for all functions that *size* depends on. But this is the empty set, thus the call introduces no dependencies, and everything is fine, i.e., type correct.

Intuitively, the type of the elements does not matter anywhere in the computation of the size of the data structure (at least, according to the way we defined that size). We can generate a component $\mathsf{cp}(\mathit{size}, \mathrm{Tree})$ and specialize the recursive call without need for an additional argument:

$$\mathsf{cp}(\mathit{size}, \mathrm{Tree})\ (\mathit{Node}\ \ell\ x\ r) = \mathsf{cp}(\mathit{size}, \mathrm{Tree})\ \ell + 1 + \mathsf{cp}(\mathit{size}, \mathrm{Tree})\ r\ \ .$$

(In Section 8.2, we will learn about a better way to define *size*, as a generic function that *does* depend on itself.)

The fact that function *size* is restricted to type arguments constructed with type constructors of kind $* \rightarrow *$ stems from its arms, not from its type. Consequently, a call to $\mathit{size}\ \langle\mathrm{Int}\rangle$ results in a specialization error rather than a type error. From the type signature of *size*, which is

$$\mathit{size}\ \langle a :: *\rangle :: a \rightarrow \mathrm{Int}\ ,$$

we can derive example types for applications of *size* to different type arguments, just as we in Figure 5.1 for *add*. This time, there are no dependencies, therefore the resulting types are simpler. The examples are shown in Figure 5.2. These types can once more be calculated automatically from the type signature using the gapp algorithm from Section 6.3.

$$\begin{array}{lllll}
\mathit{size}\ \langle A :: *\rangle & :: & A & \rightarrow \mathrm{Int} \\
\mathit{size}\ \langle A\ (\alpha :: *) :: *\rangle & :: \forall a :: *. & A\ a & \rightarrow \mathrm{Int} \\
\mathit{size}\ \langle A\ (\alpha :: *)\ (\beta :: *) :: *\rangle & :: \forall (a :: *)\ (b :: *). & A\ a\ b & \rightarrow \mathrm{Int} \\
\mathit{size}\ \langle A\ (\alpha :: * \rightarrow *) :: *\rangle & :: \forall (a :: * \rightarrow *). & A\ a & \rightarrow \mathrm{Int}\ .
\end{array}$$

Figure 5.2: Types for generic applications of *size* to type arguments of different form

It is perfectly possible to define an arm of *size* for type patterns involving type constructors of different kind such as pairs

$$\mathit{size}\ \langle(\alpha, \beta)\rangle = \mathit{const}\ 2\ ,$$

or integers

$$\mathit{size}\ \langle\mathrm{Int}\rangle \quad = \mathit{const}\ 0\ \ .$$

Next, let us look at an example of a function that depends on another function. Suppose we want to define a partial order that works on lists and pairs and combinations thereof. We define two lists to be comparable only if they both have the same size and the elements are comparable pointwise. If the comparison yields the same result for all elements, then that is the result of the function. For pairs though, two elements are only comparable if the first components are *equal*, the result being the result of comparing the second components:

```
data CprResult = Less | More | Equal | NotComparable
cpr ⟨[α]⟩    x        y
  | size ⟨[α]⟩ x == size ⟨[α]⟩ y = if size ⟨[α]⟩ x == 0
                                  then Equal
                                  else let p = zipWith (cpr ⟨α⟩) x y
                                       in  if allEqual ⟨[CprResult]⟩ p
                                             then head p
                                             else NotComparable
  | otherwise                    = NotComparable
cpr ⟨(α, β)⟩ (x₁, x₂) (y₁, y₂)
  | equal ⟨α⟩ x₁ y₁              = cpr ⟨β⟩ x₂ y₂
  | otherwise                    = NotComparable .
```

This function deliberately uses quite a number of other type-indexed functions which we assume to be defined: *size* we already know, *equal* is a function to test two values for equality, and *allEqual* is a function to check for a list (and possibly other data structures) whether all elements stored in the list are equal. Finally, *cpr* itself is used recursively. We assume that the type signatures for *equal* and *allEqual* are as follows:

$$\begin{aligned} &\mathit{equal}\ \langle a :: *\rangle :: (\mathit{equal}) \Rightarrow a \to a \to \mathrm{Bool} \\ &\mathit{allEqual}\ \langle a :: *\rangle :: (\mathit{equal}) \Rightarrow a \to \mathrm{Bool} \end{aligned}$$

Whereas *equal* is supposed to test two values of the same type for equality, the function *allEqual* is intended to be used on data structures – such as a list, [CprResult], in the definition of *cpr* – to test if all elements of that data structure are equal.

The question is: what is the type signature for *cpr*? You may try to guess the answer before reading on, as an exercise for your intuition.

The situation of the calls to *size* are as in the Tree arm of *size* itself: no dependency is needed. The type argument $[\alpha]$ does contain a dependency variable, but not in the head. Therefore, because *size* does not depend on itself, the specialization to the List arm can be made without referring to the element type α.

Even more clearly, there is no dependency on *allEqual*. The call has a constant type argument, and the specialization to constant type arguments is always possible without additional arguments and hence does never cause a dependency.

The function *equal* is a different story: in the arm for pairs, we call *equal* $\langle\alpha\rangle$, therefore there *is* a dependency on *equal*. And, because *cpr* is called on α in the arm for lists, and on β in the arm for pairs, there is also a dependency on *cpr*. Hence, the type signature of *cpr* is

$$cpr\ \langle a :: *\rangle :: (equal, cpr) \Rightarrow a \to a \to \text{CprResult}\ \ .$$

Note that the type signature is not as fine-grained as one might expect: a dependency is a global property of a type-indexed function, not attached to some arm. Although the arm for lists does not depend on *equal*, the type for *cpr* $\langle[\alpha]\rangle$ that is derived from the above type signature exhibits the dependency:

$$\begin{aligned}cpr\ \langle[\alpha]\rangle :: \forall a :: *.\ (&equal\ \langle\alpha\rangle :: a \to a \to \text{Bool},\\ &cpr\ \ \ \langle\alpha\rangle :: a \to a \to \text{CprResult})\\ &\ \ \Rightarrow [a] \to [a] \to \text{CprResult}\ \ .\end{aligned}$$

Also, there is no distinction between different dependency variables. The arm for pairs does only depend on *cpr* for the second component. Nevertheless, the type for *cpr* $\langle\alpha,\beta\rangle$ contains the dependency on *cpr* for both components:

$$\begin{aligned}cpr\ \langle(\alpha,\beta)\rangle :: \forall(a :: *)\ (b :: *).\ (&equal\ \langle\alpha\rangle :: a \to a \to \text{Bool},\\ &cpr\ \ \ \langle\alpha\rangle :: a \to a \to \text{CprResult},\\ &equal\ \langle\beta\rangle :: b \to b \to \text{Bool},\\ &cpr\ \ \ \langle\beta\rangle :: b \to b \to \text{CprResult})\\ &\ \ \Rightarrow (a,b) \to (a,b) \to \text{CprResult}\ \ .\end{aligned}$$

Another look at type classes reveals again a certain connection, this time between dependencies and *instance rules*, i.e., instances that have a condition. If we had a type class with a method *size*, such as

```
class    Size a        where
  size             :: a → Int ,
```

we could define instances for lists and trees as follows:

```
instance Size [a]        where
  size x           = length x
instance Size (Tree a) where
  size Leaf        = 0
  size (Node ℓ x r) = size ℓ + 1 + size r  .
```

The instance definitions are universal in the sense that they work for *all* lists and *all* trees, without any condition on the element type.

However, if the class for the *add* function, defined in Section 4.2 as

```
class          Add a   where
  add                  :: a → a → a
```

needed to be extended to lists in the same way as the type-indexed function, we must write

```
instance Add a ⇒ Add [a] where
  add x y
    | length x == length y = map (uncurry add) (zip x y)
    | otherwise            = error "args must have same length" .
```

The fact that the type-indexed function depends on itself is mirrored in the constraint on the instance declaration: an Add instance for $[a]$ can only be defined if a is an instance of class Add as well.

5.4 Type application in type arguments

Assuming we have successfully defined a type-indexed function such as *add* to work on lists, it is only natural that we also want to use it somewhere. In Chapter 4, a generic application had the form $x\ \langle T \rangle$, i.e., the language of type arguments was restricted to named types.

We are going to extend the syntax for *type arguments* (as opposed to type patterns) to include type application. We have already used such type arguments in the examples in the previous sections; for instance, the definition of *size* on page 53 contains the call *size* $\langle \text{Tree}\ \alpha \rangle$ on the right hand side of the case for trees.

Perhaps more interesting, the function *cpr* uses the call *allEqual* $\langle [\text{CprResult}] \rangle$, thus an application of two named types to each other.

A comparable situation would be a call to *add* $\langle [\text{Int}] \rangle$. For example, the call

$$\mathit{add}\ \langle [\text{Int}] \rangle\ [1,2,3]\ [2,3,5]$$

should have type $[\text{Int}]$ and evaluate to $[3,5,8]$. The question is, how is *add* specialized to $[\text{Int}]$? The answer is that an application in the type argument is translated into an application of specializations. There is a definition for *add* $\langle [\alpha] \rangle$ – we thus know how to add two lists, provided that we know how to add list elements, i.e., that we have access to *add* $\langle \alpha \rangle$. We have also defined the case *add* $\langle \text{Int} \rangle$, therefore we also know how to add two integers. It is pretty obvious that these two cases can be combined to get to *add* $\langle [\text{Int}] \rangle$, by supplying *add* $\langle \text{Int} \rangle$ for the dependency on *add* $\langle \alpha \rangle$ that is required by *add* $\langle [\alpha] \rangle$.

In the translation, dependencies are turned into explicit function arguments. It has already been sketched that the component for *add* on [ ] looks as follows:

```
cp(add, [ ]) cp(add, α) x y
  | length x == length y = map (uncurry cp(add, α)) (zip x y)
  | otherwise            = error "args must have same length" .
```

The dependency of *add* on itself is translated into an explicit additional argument. This can now be supplied using the component of *add* for Int, such that *add* ⟨[Int]⟩ can be specialized to

```
cp(add, [ ]) cp(add, Int) .
```

This is the general idea: dependencies reappear in the translation as explicit arguments. If a type-indexed function is called with a complex type argument that contains type application, the translation process automatically fills in the required dependencies. Thus, a call to *cpr* ⟨[Int]⟩ requires *two* dependencies to be supplied and is translated to

```
cp(cpr, [ ]) cp(equal, Int) cp(cpr, Int) ,
```

under the precondition that both *equal* and *cpr* have arms for Int. If that is not the case, such as in our definition of *cpr* above, a specialization error is be reported.

With the help of the dependencies in the types of type-indexed functions, we can prove the translation technique sketched above correct. In other words, specializing a generic application to an application of specializations will always produce a type correct term in FCR, and the type of that term is related to the dependency type the term has in the original language that supports type-indexed functions. The proof will be given in Section 6.5. For the programmer, this correctness means that type-indexed functions can be used "naturally", even for complex type arguments, because all plugging and plumbing happens under the hood.

The translation method just outlined also implies a couple of restrictions that should hold for dependencies. If, for instance, the call *add* ⟨[Int]⟩ is translated to cp(*add*, [ ]) cp(*add*, Int), then the type of cp(*add*, Int) must match the the type of the argument expected by cp(*add*, [ ]). This explains why, in the examples above, the types that appear in the dependency constraints are always related to the base type of the function in question.

Furthermore, the dependency relation is transitive. Assume that function x depends on both x and y, and function y depends on both y and z. If T_1 and T_2 are two named types, and the call $x\ \langle T_1\ (T_2\ \alpha)\rangle$ occurs on the right hand side of the definition of an arm of x, this call would be – incorrectly, as we shall see below – translated to

$$\begin{aligned}\mathsf{cp}(x,T_1)\ &\bigl(\mathsf{cp}(x,T_2)\ \mathsf{cp}(x,\alpha)\ \mathsf{cp}(y,\alpha)\bigr)\\ &\bigl(\mathsf{cp}(y,T_2)\ \mathsf{cp}(y,\alpha)\ \mathsf{cp}(z,\alpha)\bigr)\ .\end{aligned}$$

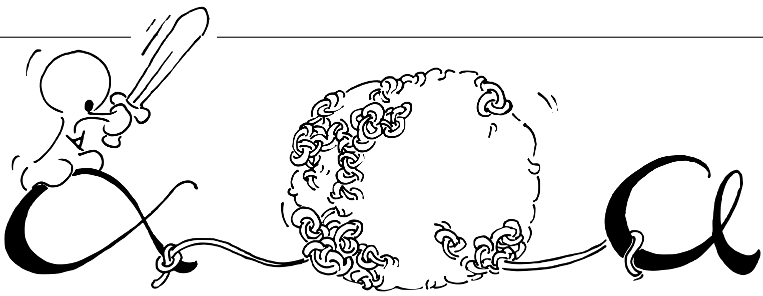
The type argument is an application of T_1 to $T_2\ \alpha$, therefore the translation is an application of the component of x for T_1 to the dependencies of x on the argument. The function x has two dependencies, x and y, thus $\mathsf{cp}(x,T_1)$ has two arguments, corresponding to the calls $x\ \langle T_2\ \alpha\rangle$ and $y\ \langle T_2\ \alpha\rangle$. In both cases the type argument is again an application, and in both cases the function that is called has again two dependencies, therefore both arguments are itself applications of a component, each to two arguments. But the function y depends on z, therefore the call to $\mathsf{cp}(z,\alpha)$ occurs in the translation! This means that the compiler must provide not only $\mathsf{cp}(x,\alpha)$ and $\mathsf{cp}(y,\alpha)$, but also $\mathsf{cp}(z,\alpha)$ in this component of x. As we have said before, we choose not to have the added complexity of different dependencies depending on the arm we are defining, thus the only solution is to make x directly depend on z as well.

The price we pay for this transitivity condition is that sometimes unnecessary dependencies are carried around. The correct translation of $x\ \langle T_1\ (T_2\ \alpha)\rangle$, for example, is

$$\begin{aligned}\mathsf{cp}(x,T_1)\ &\bigl(\mathsf{cp}(x,T_2)\ \mathsf{cp}(x,\alpha)\ \mathsf{cp}(y,\alpha)\ \mathsf{cp}(z,\alpha)\bigr)\\ &\bigl(\mathsf{cp}(y,T_2)\ \mathsf{cp}(y,\alpha)\ \mathsf{cp}(z,\alpha)\bigr)\\ &\bigl(\mathsf{cp}(z,T_2)\ \mathsf{cp}(z,\alpha)\bigr)\ ,\end{aligned}$$

assuming that z depends only on itself.

6 Dependencies



This chapter constitutes a formalization of the language extensions that have been introduced in the previous Chapter 5. We will introduce a new language FCR+tif+par, based on FCR+tif, that can handle parametrized type patterns and type-indexed functions with dependencies. The syntax of the language is introduced in Section 6.1. Subsequently, we delve into the details of dependencies and discuss dependency variables in Section 6.2 and dependency types in Section 6.3. We will then explain how to translate programs in the extended language FCR+tif+par to programs in FCR, in Section 6.4. In Section 6.5, we discuss why the translation is correct. We conclude with Section 6.6, which puts the theory covered in this chapter in relation with Ralf Hinze's work on generic programming.

6.1 Core language with parametrized type patterns

We will now extend the language FCR+tif of Figure 4.1 – our functional core language with type-indexed functions – to cover the extensions necessary to handle the new forms of type patterns and type arguments, and dependencies between

Qualified types	
$q \quad ::= \{\forall a_i :: \kappa_i.\}^{i \in 1..n}\ (\Delta) \Rightarrow t$	qualified type
Constraint sets	
$\Delta \quad ::= \{Y_i\}_{,}^{i \in 1..n}$	constraint set
Type constraints	
$Y \quad ::= x\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: q$	dependency constraint
Type patterns	
$P \quad ::= T\ \{\alpha_i\}^{i \in 1..n}$	parametrized named type pattern
Type arguments	
$A \quad ::= T$	named type, from Figure 4.1
$\quad \mid\ \alpha, \beta, \gamma, \ldots$	dependency variable
$\quad \mid\ (A_1\ A_2)$	type application
Type signatures	
$\sigma \quad ::= (\{y_k\}_{,}^{k \in 1..n}) \Rightarrow t$	type signature of type-indexed function

Figure 6.1: Core language with type-indexed functions and parametrized type patterns FCR+tif+par, extends language FCR+tif in Figure 4.1

type-indexed functions. The extended language is called FCR+tif+par. The additional syntactic constructs are shown in Figure 6.1.

Dependency variables are a new form of type variables used in the context of dependencies. We distinguish them by using lowercase Greek letters (α, β, γ) for them. Dependency variables are the topic of Section 6.2.

A **qualified type** is a type with dependency constraints. A *dependency constraint* is of the form

$$x\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: q$$

and consists of the name of a type-indexed function x, a type argument consisting of dependency variables, and a qualified type. The constraint expresses a dependency on function x at dependency variable α_0. Depending on the kind of the dependency variable α_0, there may be further dependency variable parameters, with kind annotations, which are local to the constraint and scope over the qualified type q only.

Dependency constraints may appear nested in the sense that the type in a particular constraint can be qualified again. However, all constraints are *always* at the beginning of a type. Type variables may be universally quantified in a qualified type.

A set of constraints is much like an environment. For a specific combination of type-indexed function x and dependency variable α_0, there may be at most one constraint. The order of the constraints is unimportant – in other words, we treat types that differ only in the order of dependency constraints as equivalent.

To keep forthcoming algorithms deterministic, and save us from a lot of reordering hassle during the definition of the translation to FCR, we will now define a canonical ordering on dependency constraints, and can henceforth assume that qualified types appear with their constraints ordered when needed. We assume that there is a total order $<^{lex}$ on both dependency variables and expression variables (lexical order would do, but it can be any other total order). We can extend this order to dependency constraints via the equivalence

$$\begin{aligned}&x_1\ \langle\alpha_1\ \{\beta_i\}^{i\in 1..m}\rangle :: q_1 <^{lex}\ x_2\ \langle\alpha_2\ \{\gamma_j\}^{j\in 1..n}\rangle :: q_2\\ &\quad\Longleftrightarrow \alpha_1 <^{lex}\ \alpha_2 \vee (\alpha_1 \equiv \alpha_2 \wedge x_1 <^{lex}\ x_2)\ .\end{aligned}$$

The types q_1 and q_2 that appear in the constraints are irrelevant for the ordering.

If a qualified type has no dependency, we often write

$$\{\forall(a_i :: \kappa_i).\}^{i\in 1..m}\ t$$

instead of

$$\{\forall(a_i :: \kappa_i).\}^{i\in 1..m}\ () \Rightarrow t\ .$$

Similarly, every ordinary type is trivially also a qualified type.

Sometimes, we split dependency constraints and write

$$\{\forall a_i :: \kappa_i.\}^{i\in 1..m}\ (\Delta) \Rightarrow q\ ,$$

with q being still a qualified type. Now, if q is of the form

$$\{\forall a_i' :: \kappa_i.\}^{i\in (m+1)..n}\ (\Delta') \Rightarrow t\ ,$$

and if there are no clashes between bound variables or dependencies in Δ and Δ', then the first type is considered equivalent to

$$\{\forall a_i :: \kappa_i.\}^{i\in 1..n}\ (\Delta, \Delta') \Rightarrow t\ .$$

If a dependency constraint occurs in both Δ and Δ', and both are equal modulo renaming of bound variables, then the rewrite step is still possible, and Δ, Δ' denotes the constraint set where the duplicate constraint occurs only once.

Back to the syntax in Figure 6.1: the old form of type patterns is subsumed by a new one, where a named type may appear followed by an arbitrary number of dependency variables as arguments. The case where there are no dependency variables is still allowed and coincides with the old form.

The syntax for type arguments is extended: dependency variables and type application are allowed here. Note that thus the type pattern language forms a subset of the type argument language, and we will make use of that fact occasionally.

Type signatures are a list of function names followed by a type. Type signatures are used to convey the type information of a type-indexed function. We discuss the syntax and semantics of type signatures further in Section 6.3.

6.2 Dependency variables and kinds

Dependency variables are a new form of variables. They do not occur in normal type expressions, but only in type patterns and type arguments – as a rule of thumb, only within the special $\langle\cdot\rangle$ parentheses.

They are completely independent of normal type variables. We use $\mathsf{fdv}(A)$ to refer to the free dependency variables of a type argument (or pattern) A, and continue to refer to the disjoint set of free type variables by $\mathsf{ftv}(A)$.

Dependency variables are bound by a type pattern. The arguments appearing in a pattern scope over the expression belonging to the branch (note that ordinary type variables never scope over expressions; they are only bound by lambdas in datatype declarations and by universal quantification). Furthermore, dependency variables can be local to a dependency constraint, if the dependency variable of the constraint is of higher kind and thus has additional parameters. Bound dependency variables can be renamed, as other sorts of bound variables, and we will assume that this is implicitly done during substitutions to prevent name capture.

The kind checking rules need to be extended to cover type arguments and type patterns, and qualified types. The form of judgment for arguments is, as for ordinary types,

$$\mathrm{K} \vdash A :: \kappa\ ,$$

stating the type argument A has kind κ under environment K. The environment K may now also contain assumptions for dependency variables, of the form $\alpha :: \kappa$. These bindings may also be contained in K during the checking of normal type expressions – they are not used there, though, only propagated.

Kind checking for type arguments makes use of the rules for ordinary types in Figure 3.2 and only needs one new judgment for dependency variables, shown in Figure 6.2. The judgment is what one would expect for a variable: we look up the variable in the environment.

$$\mathrm{K} \vdash A :: \kappa$$

$$\frac{\alpha :: \kappa \in \mathrm{K}}{\mathrm{K} \vdash \alpha :: \kappa} \quad \text{(t-depvar)}$$

Figure 6.2: Kind checking for language FCR+tif+par of Figure 6.1, extends Figure 3.2

For patterns, we use a modified judgment of the form

$$\mathrm{K}_1 \vdash^{pat} P :: \kappa \rightsquigarrow \mathrm{K}_2 \;,$$

which is analogous to the judgment for value-level patterns (cf. Figure 3.4). Under input environment K_1 we find pattern P to be of kind κ, thereby binding the dependency variables in K_2. The rules are stated in Figure 6.3. For a named type, we look up its kind in the input environment, and then bind the arguments to the appropriate kinds. The subsidiary judgment for dependency variables is trivial.

$$\mathrm{K}_1 \vdash^{pat} P :: \kappa \rightsquigarrow \mathrm{K}_2$$

$$\frac{}{\mathrm{K} \vdash^{pat} \alpha :: \kappa \rightsquigarrow \alpha :: \kappa} \quad \text{(tp-depvar)}$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash T :: \{\kappa_i \rightarrow\}^{i \in 1..n} \; * \\ \{\mathrm{K} \vdash^{pat} \alpha_i :: \kappa_i \rightsquigarrow \mathrm{K}_i\}^{i \in 1..n}\end{array}}{\mathrm{K} \vdash^{pat} T \; \{\alpha_i\}^{i \in 1..n} :: * \rightsquigarrow \{\mathrm{K}_i\}_{,}^{i \in 1..n}} \quad \text{(tp-named)}$$

Figure 6.3: Kind checking of type patterns in language FCR+tif+par of Figure 6.1

Qualified types are always of kind $*$, thus the corresponding judgment is:

$$\mathrm{K} \vdash q :: * \;.$$

The rule is given in Figure 6.5, and makes use of the judgment for well-formedness of a dependency constraint in Figure 6.4, which is of the form

$$\mathrm{K} \vdash Y :: * \;.$$

$$\mathrm{K} \vdash Y :: *$$

$$\frac{\mathrm{K}' \equiv \mathrm{K}\ \{,\alpha_i :: \kappa_i\}^{i\in 1..n} \qquad \mathrm{K}' \vdash \alpha_0\ \{\alpha_i\}^{i\in 0..n} :: * \qquad \mathrm{K}' \vdash q :: *}{\mathrm{K} \vdash \bigl(x\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i\in 1..n}\rangle :: q\bigr) :: *} \quad \text{(t-constraint)}$$

Figure 6.4: Well-formedness of dependency constraints in language FCR+tif+par of Figure 6.1

$$\mathrm{K} \vdash q :: *$$

$$\frac{\mathrm{K}' \equiv \mathrm{K}\ \{,a_i :: \kappa_i\}^{i\in 1..m} \qquad \{\mathrm{K}' \vdash Y_j :: *\}^{j\in 1..n} \qquad \mathrm{K}' \vdash t :: *}{\mathrm{K} \vdash \bigl(\{\forall a_i :: \kappa_i .\}^{i\in 1..m}\ (\{Y_j\}_{,}^{j\in 1..n}) \Rightarrow t\bigr) :: *} \quad \text{(t-qual)}$$

Figure 6.5: Kind checking of qualified types in FCR+tif+par of Figure 6.1

6.3 Dependency types

This section is a formalization of Section 5.3. We explain how qualified types can be assigned to both calls of type-indexed functions and arms of a typecase.

6.3.1 Type signatures of type-indexed functions

During type checking, we have to verify that arms of type-indexed functions and occurrences of calls to type-indexed functions are type correct. In the situation of Chapter 4, this was relatively easy: for each generic function x, we added a *type signature* of form

$$x\ \langle a :: *\rangle :: t$$

to the type environment Γ; and to compute the type of $x\ \langle T\rangle$ (or, similarly, the type of the right hand side of the arm for T in the definition of x), we just returned

$$t[T\ /\ a]\ .$$

Type arguments were not allowed if they were not of the form T, with T being a named type of kind $*$.

The situation is far more difficult now: we have more complex type patterns and arguments, and if they contain dependency variables, dependencies might arise. In Section 5.3, we have already noted that the syntax for type signatures of type-indexed functions must be extended to cover the dependencies of a function. These extended type signatures are of the form

$$x\ \langle a :: * \rangle :: (\{y_k\}_{,}^{k \in 1..n}) \Rightarrow t\ .$$

Such a type signature consists of a number of *dependencies* y_k, which are names of other type-indexed functions, and an unqualified type t, which we call the *base type* of the function. The type variable a is bound on the right hand side and may thus appear free in the dependencies and the base type. We use the symbol σ as a metavariable for a type signature.

From a type signature for a type-indexed function x – so we have demonstrated by example – we can generate qualified types for $x\ \langle A \rangle$ for many different type arguments (and, similarly, also type patterns) A.

For type checking in FCR+tif+par, the type environment Γ contains bindings of the form $x :: t$ to assign unqualified types to variables, and $x\ \langle a :: * \rangle :: \sigma$, to store type signatures of generic functions.

We use a judgment of the form

$$\mathsf{base}_{K;\Gamma}(x\ \langle A \rangle) \equiv t$$

to instantiate the base type of a generic function. The rule is given in Figure 6.6, together with a judgment

$$\mathsf{dependencies}_{\Gamma}(x) \equiv \{y_k\}_{,}^{k \in 1..n}$$

that can be applied to extract the dependencies of a type-indexed function. Both judgments can be used as functions.

The *well-formedness* of a type-indexed function's type signature is relatively easy to verify: the dependencies must be type-indexed functions in scope, and the base type must be of kind $*$. In addition, the dependency relation must be equal to its transitive closure: if one function indirectly depends on another, it also depends on it directly (cf. Section 5.4).

We express the well-formedness of a type signature formally via

$$K; \Gamma \vdash^{tpsig} x\ \langle a :: * \rangle :: \sigma\ .$$

The definition is given in Figure 6.7. This judgment is used later in rule (d/tr-tif), while type checking and translating the declaration of a type-indexed function.

$$\mathsf{base}_{K;\Gamma}(x\ \langle A \rangle) \equiv t$$

$$\frac{x\ \langle a :: * \rangle :: (\{y_k\}_{,}^{k \in 1..n}) \Rightarrow t \in \Gamma \qquad K \vdash A :: *}{\mathsf{base}_{K;\Gamma}(x\ \langle A \rangle) \equiv t[A\ /\ a]} \quad \text{(base)}$$

$$\mathsf{dependencies}_{\Gamma}(x) \equiv \{y_k\}_{,}^{k \in 1..n}$$

$$\frac{x\ \langle a :: * \rangle :: (\{y_k\}_{,}^{k \in 1..n}) \Rightarrow t \in \Gamma}{\mathsf{dependencies}_{\Gamma}(x) \equiv \{y_k\}_{,}^{k \in 1..n}} \quad \text{(deps)}$$

Figure 6.6: Extracting information from the type signature of a type-indexed function

$$K; \Gamma \vdash^{tpsig} x\ \langle a :: * \rangle :: \sigma$$

$$\frac{\begin{array}{c} \left\{K; \Gamma \vdash^{tpsig} y_k\ \langle a :: * \rangle :: (\{z_{k,i}\}_{,}^{i \in 1..m_k}) \Rightarrow t_k\right\}^{k \in 1..n} \\ \left\{\{z_{k,i} \in \{y_j\}_{,}^{j \in 1..n}\}^{i \in 1..m_k}\right\}^{k \in 1..n} \\ K, a :: * \vdash t :: * \end{array}}{K; \Gamma \vdash^{tpsig} x\ \langle a :: * \rangle :: (\{y_k\}_{,}^{k \in 1..n}) \Rightarrow t} \quad \text{(typesig)}$$

Figure 6.7: Well-formedness of type signatures for type-indexed functions

6.3.2 Generic application algorithm

Finally, we have collected all the prerequisites to start defining an algorithm that can, from a type signature for a type-indexed function x, compute the (qualified) type of $x\ \langle A\rangle$ for some type argument A. Judgments for this algorithm are of the form

$$\mathsf{gapp}_{K;\Gamma}(x\ \langle A\rangle) \equiv q\ .$$

The algorithm is shown in Figure 6.8 and makes use of a subsidiary judgment to create a single dependency constraint, of the form

$$\mathsf{mkdep}_{K;\Gamma}(x\ \langle \alpha \leftarrow a\rangle) \equiv Y\ .$$

The idea of the algorithm is that for each free dependency variable that occurs in the type argument A of function x, one set of dependencies is introduced. Rule (ga-1) is thus for type arguments which do not contain any free dependency variables. If a dependency variable is contained in A, we distinguish two cases. If the dependency variable appears in the head of A, then (ga-2) applies, otherwise (ga-3).

If there are no free dependency variables in A, (ga-1) states that the desired type is the instantiated base type of x.

Both (ga-2) and (ga-3) deal with the occurrence of a dependency variable α in A. In both rules dependencies for α are introduced, and in both rules we create a fresh type variable a to replace α, which should be of the same kind as α. This substitution takes place because we do not allow dependency variables to appear anywhere but in type arguments, and because later, in the generalization that will be discussed in Chapter 9, one dependency variable can be instantiated to multiple type variables.

If α occurs in the head of A, then a dependency of x on α is introduced in (ga-2), implementing the case – discussed in Section 5.3 – that a call to a function x where α is the head of the type argument does *always* introduce a dependency of x on α. The dependency constraint itself is generated by the call $\mathsf{mkdep}_x(y\ \langle \alpha \leftarrow a\rangle)$, explained below. The remainder of the qualified type is then generated by the recursive call to gapp, where the head occurrence of α in the type argument has been replaced by a.

If α occurs in A, but not in the head, then dependencies are introduced for all functions y_k that x depends on. This time, in the recursive call to gapp, all occurrences of α are substituted by a.

Note that if α occurs in both the head of the type argument A and somewhere else in A, then first rule (ga-2) and then rule (ga-3) is applied, creating dependencies for x on α as well as for all dependencies y_k on α in the process. If x depends

$$\mathsf{gapp}_{\mathrm{K};\Gamma}(x\ \langle A\rangle) \equiv q$$

$$\frac{\mathrm{K} \vdash A :: * \qquad \mathsf{fdv}(A) \equiv \varepsilon}{\mathsf{gapp}_{\mathrm{K};\Gamma}(x\ \langle A\rangle) \equiv \mathsf{base}_{\mathrm{K};\Gamma}(x\ \langle A\rangle)} \quad \text{(ga-1)}$$

$$\frac{\mathrm{K} \vdash \alpha :: \kappa \qquad a \text{ fresh} \qquad \mathrm{K}' \equiv \mathrm{K}, a :: \kappa}{\begin{array}{l}\mathsf{gapp}_{\mathrm{K};\Gamma}(x\ \langle \alpha\ \{A_i\}^{i\in 1..n}\rangle) \equiv \forall a :: \kappa.\ (\mathsf{mkdep}_{\mathrm{K}';\Gamma}(x\ \langle \alpha \leftarrow a\rangle)) \\ \qquad\qquad\qquad\qquad \Rightarrow \mathsf{gapp}_{\mathrm{K}';\Gamma}(x\ \langle a\ \{A_i\}^{i\in 1..n}\rangle)\end{array}} \quad \text{(ga-2)}$$

$$\frac{\begin{array}{c}\alpha \in \mathsf{fdv}(A) \qquad \mathsf{head}(A) \not\equiv \alpha \\ \mathrm{K} \vdash \alpha :: \kappa \qquad a \text{ fresh} \qquad \mathrm{K}' \equiv \mathrm{K}, a :: \kappa \\ \mathsf{dependencies}_{\Gamma}(x) \equiv \{y_k\}_{,}^{k\in 1..n}\end{array}}{\begin{array}{l}\mathsf{gapp}_{\mathrm{K};\Gamma}(x\ \langle A\rangle) \equiv \forall a :: \kappa.\ (\{\mathsf{mkdep}_{\mathrm{K}';\Gamma}(y_k\ \langle \alpha \leftarrow a\rangle)\}_{,}^{k\in 1..n}) \\ \qquad\qquad\qquad\qquad \Rightarrow \mathsf{gapp}_{\mathrm{K}';\Gamma}(x\ \langle A[a\ /\ \alpha]\rangle)\end{array}} \quad \text{(ga-3)}$$

$$\mathsf{mkdep}_{\mathrm{K};\Gamma}(y\ \langle \alpha \leftarrow a\rangle) \equiv Y$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash \alpha :: \kappa \qquad \kappa \equiv \{\kappa_h \rightarrow\}^{h\in 1..\ell}\ * \\ \{\gamma_h \text{ fresh}\}^{h\in 1..\ell} \qquad \mathrm{K}' \equiv \mathrm{K}\ \{, \gamma_h :: \kappa_h\}^{h\in 1..\ell}\end{array}}{\begin{array}{l}\mathsf{mkdep}_{\mathrm{K};\Gamma}(x\ \langle \alpha \leftarrow a\rangle) \\ \quad \equiv (x\ \langle \alpha\ \{\gamma_h\}^{h\in 1..\ell}\rangle :: \mathsf{gapp}_{\mathrm{K};\Gamma}(x\ \langle a\ \{\gamma_h\}^{h\in 1..\ell}\rangle))\end{array}} \quad \text{(mkdep)}$$

Figure 6.8: Generic application algorithm

on itself, the algorithm generates a double dependency constraint with the same type, in which case we ignore the duplicate and just keep a single constraint.

The algorithm gapp is nondeterministic, because it leaves the choice of which free dependency variable in the type argument to choose next. For the result, the order makes no difference; only the dependency constraints of the resulting type will be generated in a different order, and the order of dependency constraints does not matter. Thus, we can assume that always the smallest dependency variable is chosen in rules (ga-2) and (ga-3), which eliminates the choice and makes gapp a deterministic algorithm that can be used as a function.

It remains to discuss the mkdep judgment, which is of the form

$$\mathsf{mkdep}_{K;\Gamma}(x\ \langle \alpha \leftarrow a \rangle) \equiv Y\ \ .$$

It is used to determine the type that is associated with a dependency on function x for the dependency variable α, which is instantiated with type variable a, where both α and a are of kind κ. The resulting constraint is Y.

There is only one rule, named (mkdep). We can almost directly apply gapp on x again to determine the type of the dependency, but α is not necessarily of kind $*$, and if it is not, local dependency variables γ_h are introduced as arguments to α. While the name of the dependency constraint makes use of the dependency variable α, the type itself is generated with α substituted by the type variable a.

Theorem 6.1 (Termination of gapp). If $K \vdash A :: *$ and x is a type-indexed function, thus

$$x\ \langle a :: * \rangle :: \sigma \in \Gamma$$

and

$$K; \Gamma \vdash^{tpsig} x\ \langle a :: * \rangle :: \sigma\ ,$$

then $\mathsf{gapp}_{K;\Gamma}(x\ \ \langle A \rangle)$ has a solution q that can be derived in a finite number of steps.

Proof. The conditions of the theorem ensure that one of the rules is always applicable.

The **size** of a kind can be defined as follows:

$$\begin{aligned} &\mathsf{size}(*) &&\equiv 1 \\ &\mathsf{size}(\kappa_1 \to \kappa_2) &&\equiv \mathsf{size}(\kappa_1) + \mathsf{size}(\kappa_2)\ \ . \end{aligned}$$

We can now compute the size of a type argument by determining all free dependency variables in it, and computing the sum of the sizes of their kinds. We count multiple occurrences of one dependency variable multiple times.

Then, in each recursive call of gapp, the size of the type argument is strictly smaller than before. In rules (ga-2) and (ga-3), we eliminate at least one dependency variable, and the size of its kind is strictly positive. In (mkdep), we know that the size of the original type argument was at least the size of κ, the kind of α (because α occurs in the original type argument). By construction,

$$\mathsf{size}(a\ \{\gamma_h\}^{h\in 1..p}) \equiv \mathsf{size}(\kappa) - 1\ .$$

Since the size of the initial type argument is finite, and the size is always a natural number, the algorithm terminates. □

For a concrete function such as *cpr*, let us investigate what the algorithm returns as type for a generic application to $T\ \alpha$, where T is some constant type constructor of kind $* \to *$, and $\alpha :: *$ is a dependency variable. The resulting type is

$$\begin{aligned} &cpr\ \langle T\ \alpha\rangle :: \forall a :: *.\ (equal\ \langle\alpha\rangle :: a \to a \to \mathrm{Bool},\\ &\qquad\qquad\qquad\qquad\ \ cpr\quad \langle\alpha\rangle :: a \to a \to \mathrm{CprResult})\\ &\qquad\qquad\qquad\qquad\quad \Rightarrow T\ a \to T\ a \to \mathrm{CprResult}\ . \end{aligned}$$

If we apply *cpr* to a type argument containing two dependency variables, such as $T\ \alpha\ \beta$, with $T :: * \to * \to *$, and both α and β of kind $*$, we obtain two sets of dependencies, because the recursive call in (ga-3) on $x\ \langle A[a\ /\ \alpha]\rangle$, after substituting one dependency variable, still contains another:

$$\begin{aligned} &cpr\ \langle T\ \alpha\ \beta\rangle :: \forall (a :: *)\ (b :: *).\ (equal\ \langle\alpha\rangle :: a \to a \to \mathrm{Bool},\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad cpr\quad \langle\alpha\rangle :: a \to a \to \mathrm{CprResult},\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad equal\ \langle\beta\rangle :: b \to b \to \mathrm{Bool},\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad cpr\quad \langle\beta\rangle :: b \to b \to \mathrm{CprResult})\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow T\ a\ b \to T\ a\ b \to \mathrm{CprResult}\ . \end{aligned}$$

If a dependency variable is itself of higher kind, such as in *cpr* $\langle T\ \alpha\rangle$, where $T :: (* \to *) \to *$ and $\alpha :: * \to *$, the result is a nested dependency constraint:

$$\begin{aligned} &cpr\ \langle T\ \alpha\rangle ::\\ &\quad \forall a :: * \to *.\ \big(equal\ \langle\alpha\ (\gamma :: *)\rangle :: \forall c :: *.\ (equal\ \langle\gamma\rangle :: c \to c \to \mathrm{Bool})\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow a\ c \to a\ c \to \mathrm{Bool},\\ &\qquad\qquad\quad cpr\quad \langle\alpha\ (\gamma :: *)\rangle :: \forall c :: *.\ (equal\ \langle\gamma\rangle :: c \to c \to \mathrm{Bool},\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad cpr\quad \langle\gamma\rangle :: c \to c \to \mathrm{CprResult})\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow a\ c \to a\ c \to \mathrm{CprResult}\big)\\ &\qquad\qquad\quad \Rightarrow T\ a \to T\ a \to \mathrm{CprResult}\ . \end{aligned}$$

Note that the overall structure is still that we have one set of dependency constraints, for the dependencies *equal* and *cpr*, on dependency variable α. But, because $\alpha :: * \rightarrow *$, each of the types of the dependencies has a dependency again, on a local dependency variable $\gamma :: *$. The type of the dependency *equal* $\langle \alpha\ (\gamma :: *) \rangle$ is exactly the same as the type for *equal* $\langle a\ \gamma \rangle$ would be (and similar for *cpr* $\langle \alpha\ (\gamma :: *) \rangle$. This stems from the fact that in (mkdep), for the dependencies the generic application algorithm is called recursively as well, with the freshly introduced type variable as argument, applied to equally freshly introduced dependency variables. You may want to refer back to Section 5.3 and confirm that the example types given for *add* and *size* in Figure 5.1 and 5.2 can all be generated with help of the gapp algorithm.

6.4 Types and translation

We will now discuss the translation of FCR+tif+par to FCR. First, we will have a close look at how types are translated: we use qualified types for arms of type-indexed functions and calls to type-indexed functions. In FCR, these qualified types are represented as ordinary functions. Section 6.4.1 shows the details. In the following Section 6.4.2, we sketch which environment we need for type checking and translating FCR+tif+par. Section 6.4.3 contains more about qualified types, before Sections 6.4.4 and 6.4.5 talk about the actual translation of expressions and declarations, respectively. In Section 6.4.6, we show how FCR+tif+par environments can be mapped to FCR environments, and in Section 6.4.7, we discuss the critical question of how to translate generic applications, i.e., calls to type-indexed functions.

6.4.1 Translation of qualified types

In the translation of FCR+tif+par to FCR, we represent values of types with dependency constraints as functions, where the constraints are represented as explicit arguments. The translation has to ensure that the dependency arguments that are needed are also provided.

Dependency constraints may be reordered, whereas function arguments usually may not. To save us from a lot of tedious and – even worse – confusing reordering work, we make use of the canonical ordering $<^{lex}$ defined in Section 6.1.

Figure 6.9 shows how such a canonically ordered qualified type can be translated into an FCR type. The judgments are of the form

$$[\![q_{\text{FCR+tif+par}}]\!]^{\text{par}} \equiv t_{\text{FCR}}$$

or

$$[\![\Delta_{\text{FCR+tif+par}}]\!]^{\text{par}} \equiv t_{\text{FCR}}[\bullet]\ ,$$

the former taking a qualified type in FCR+tif+par to a type in FCR, the latter taking dependency constraints to a type with a hole. We use the notation $t[\bullet]$ to denote a type where one position that syntactically could hold another type has been replaced by a hole, written $\bullet$. If u is another type, then $t[u]$ is the type where the hole is filled by type u.

As an example of how the translation proceeds, recall the type for $cpr\ \langle T\ \alpha\rangle$ with $T :: (* \to *) \to *$ from page 80. Rewritten in canonical order, this type is

$$\begin{aligned}\forall a :: * \to *.\ (&cpr\ \ \ \langle \alpha\ (\gamma :: *)\rangle :: \forall c :: *.\ (cpr\ \ \ \langle\gamma\rangle :: c \to c \to \text{CprResult},\\ &\qquad\qquad\qquad\qquad\qquad\quad equal\ \langle\gamma\rangle :: c \to c \to \text{Bool})\\ &\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow a\ c \to a\ c \to \text{CprResult},\\ &equal\ \langle \alpha\ (\gamma :: *)\rangle :: \forall c :: *.\ (equal\ \langle\gamma\rangle :: c \to c \to \text{Bool})\\ &\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow a\ c \to a\ c \to \text{Bool})\\ &\Rightarrow T\ a \to T\ a \to \text{CprResult}\ .\end{aligned}$$

The translation of this type, according to the rules in Figure 6.9, is

$$\begin{aligned}\forall a :: * \to *.\ \ &(\forall c :: *.\ \ (c \to c \to \text{CprResult})\\ &\qquad\qquad \to (c \to c \to \text{Bool})\\ &\qquad\qquad \to a\ c \to a\ c \to \text{CprResult})\\ \to\ &(\forall c :: *.\ \ (c \to c \to \text{Bool})\\ &\qquad\qquad \to a\ c \to a\ c \to \text{Bool})\\ \to\ &T\ a \to T\ a \to \text{CprResult}\ .\end{aligned}$$

In the translation, one can no longer see with which function and dependency variable a certain argument is associated. But because we have agreed on a canonical ordering of the constraints, the position of the function argument is enough to recover that information. Furthermore, nested dependency constraints translate to higher-order functions.

A simpler example would be $add\ \langle[\alpha]\rangle$. Here, the qualified type is

$$\forall a.(add\ \langle\alpha\rangle :: a \to a \to a) \Rightarrow [a] \to [a] \to [a]\ ,$$

and the corresponding FCR type is simply

$$\forall a.(a \to a \to a) \to [a] \to [a] \to [a]\ \ .$$

6.4.2 Environments for type checking and translation

Because the presence of dependency constraints plays an important role for the result of the translation, we combine type checking and translation rules for the

$$[\![q_{\textsc{fcr}+\text{tif}+\text{par}}]\!]^{\text{par}} \equiv t_{\textsc{fcr}}$$

$$\frac{[\![\Delta]\!]^{\text{par}} \equiv u[\bullet]}{[\![\{\forall a_i :: \kappa_i.\}^{i\in 1..n}\ (\Delta) \Rightarrow t]\!]^{\text{par}} \equiv \{\forall a_i :: \kappa_i.\}^{i\in 1..n}\ u[t]} \quad \text{(tr-qtype)}$$

$$[\![\Delta_{\textsc{fcr}+\text{tif}+\text{par}}]\!]^{\text{par}} \equiv t_{\textsc{fcr}}[\bullet]$$

$$\frac{}{[\![\varepsilon]\!]^{\text{par}} \equiv \bullet} \quad \text{(tr-constraint-1)}$$

$$\frac{\begin{array}{c}[\![\Delta]\!]^{\text{par}} \equiv u[\bullet] \\ [\![q]\!]^{\text{par}} \equiv t\end{array}}{[\![\Delta, x\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i\in 1..n}\rangle :: q]\!]^{\text{par}} \equiv u[t \to \bullet]} \quad \text{(tr-constraint-2)}$$

Figure 6.9: Translation of qualified types and dependency constraints in language FCR+tif+par

language FCR+tif+par. In this process, we make use of four environments. The kind environment K stores kinds for both type and dependency variables.

The type environment Γ stores the types of variables and type signatures of type-indexed functions. Again, we make use of a signature environment Σ, in which we store the signatures of type-indexed functions that are in scope. Note that we distinguish signatures of type-indexed functions, such as defined in Section 4.4, and type signatures. The former contain a list of types and type constructors that appear in the type patterns of the **typecase** construct. The latter contain information on how to type check calls to a type-indexed functions.

New is a **dependency environment** Δ. We use the same symbol that we use to denote a dependency constraint set, because both are of the same form: the dependency environment Δ contains bindings of the form

$$x\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: q$$

and stores dependencies that are currently available. If a dependency is available, that means that in the translation, a variable $\mathsf{cp}(x, \alpha_0)$ of type $[\![q]\!]^{\mathrm{par}}$ is in scope.

6.4.3 Coercing qualified types

Given a dependency environment Δ, it is possible to coerce a value of a certain qualified type into a value of another qualified type, either adding or removing dependency constraints in the process. The idea is that on the level of the language FCR+tif+par, we have a subtyping relation between two qualified types q_1 and q_2, if on the level of the language FCR, we can find a coercion function that transforms any value of type $[\![q_1]\!]^{\mathrm{par}}$ into a value of type $[\![q_2]\!]^{\mathrm{par}}$.

Following this idea, we have rules of the form

$$\mathrm{K}; \Delta \vdash q_1 \leqslant q_2 \rightsquigarrow e[\bullet]\ ,$$

that are given in Figure 6.10 and extend the subsumption relation on FCR types in Figure 3.5. In some cases, when we are not interested in the coercion term, we write

$$\mathrm{K}; \Delta \vdash q_1 \leqslant q_2$$

for simplicity.

A subsumption judgment is to be interpreted as the following statement: under environments K and Δ, any expression $e_{\mathrm{FCR+tif+par}}$ of type q_1 can be used as a value of type q_2. But the types $[\![q_1]\!]^{\mathrm{par}}$ and $[\![q_2]\!]^{\mathrm{par}}$ are different in FCR. Therefore, the translation of $e_{\mathrm{FCR+tif+par}}$ must be different depending on whether it is given type q_1 or type q_2. The coercion function $e[\bullet]$ – which is a holed expression in the

$$K; \Delta \vdash q_1 \leqslant q_2 \rightsquigarrow e[\bullet]$$

$$\frac{\Delta_1, \Delta_2 \vdash q \leqslant t \rightsquigarrow e[\bullet] \qquad \vdash \Delta_2 \rightsquigarrow e_\Delta[\bullet]}{\Delta_1 \vdash q \leqslant (\Delta_2) \Rightarrow t \rightsquigarrow e_\Delta[e[\bullet]]} \quad \text{(qs-dep-1)}$$

$$\frac{K \vdash t_1 \leqslant t_2 \qquad K; \Delta_1 \Vdash \Delta_2 \rightsquigarrow e[\bullet]}{K; \Delta_1 \vdash (\Delta_2) \Rightarrow t_1 \leqslant t_2 \rightsquigarrow e[\bullet]} \quad \text{(qs-dep-2)}$$

$$\frac{\begin{array}{c} a \notin \mathsf{ftv}(q_1) \\ K, a :: \kappa; \Delta \vdash q_1 \leqslant q_2 \rightsquigarrow e[\bullet] \end{array}}{K; \Delta \vdash q_1 \leqslant \forall a :: \kappa.\ q_2 \rightsquigarrow e[\bullet]} \quad \text{(qs-skol)}$$

$$\frac{\begin{array}{c} K \vdash u :: \kappa \\ K; \Delta \vdash q_1[u \mathbin{/} a] \leqslant q_2 \rightsquigarrow e[\bullet] \end{array}}{K; \Delta \vdash \forall a :: \kappa.\ q_1 \leqslant q_2 \rightsquigarrow e[\bullet]} \quad \text{(qs-inst)}$$

Figure 6.10: Subsumption relation on qualified types, extends Figure 3.5

same way as we had holed types before – can be used to modify the translation accordingly.

If the subsumption relation is used as an algorithm to determine a coercion function, there is a strategy: first the rules (qs-skol) and (qs-inst) can be applied to "remove" universal quantifiers from the types. Both are analogues of the rules (s-skol) and (s-inst) that deal with universal quantification in unqualified types.

After the quantifiers have been dealt with, rule (qs-dep-1) allows to add the constraints Δ_2 of the right qualified type $(\Delta_2) \Rightarrow t$ to the environment of assumed constraints Δ_1. We then check the left qualified type q to be convertible in t under the extended environment Δ_1, Δ_2. The conversion is witnessed by $e[\bullet]$, but this expression context may make use of the dependencies in Δ_1 and Δ_2. To turn this into an expression that only makes use of the dependencies in Δ_1, we have to abstract from Δ_2 again. This is the job of the judgment $\vdash \Delta_2 \rightsquigarrow e_\Delta[\bullet]$.

The subsidiary judgment form

$$\vdash \Delta \rightsquigarrow e[\bullet]$$

is defined in Figure 6.11. It maps a list of dependency constraints Δ into a sequence of lambda abstractions of the associated component names.

Rule (qs-dep-2) of Figure 6.10 is for the case that the right hand side dependencies have been eliminated with help of (qs-dep-1), but the left-hand side still is a qualified type $(\Delta_2) \Rightarrow t_1$. In this case we check that $\vdash t_1 \leqslant t_2$ using the subsumption relation on normal types from Figure 3.5. Ordinary types are FCR types, therefore we do not need a coercion function witnessing the coercion from t_1 to t_2. But the dependencies in Δ_2 express arguments that need to be provided, and we must verify that environment Δ_1 contains enough information to provide these arguments. This is achieved using the entailment judgment $\Delta_1 \Vdash \Delta_2 \rightsquigarrow e[\bullet]$, which delivers the expression context that is returned.

The entailment judgment form

$$\mathrm{K}; \Delta_1 \Vdash \Delta_2 \rightsquigarrow e[\bullet]$$

is given in Figure 6.12. It checks if the dependencies available in Δ_1 are sufficient to supply the dependencies required by Δ_2 and returns a number of applications to the appropriate dependency arguments as a witness in the expression context $e[\bullet]$. Rule (ent-2) is the important case: if $x\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: q_2$ is the first of the expected constraints (according to the canonical ordering of the dependency constraints), there must be a constraint for x on α among the available constraints as well, such as $x\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: q_1$. We recursively satisfy the remaining dependencies Δ_2 by invoking the entailment judgment again, yielding an expression context e_Δ that can supply all the dependencies in Δ_2. It remains to supply the dependency on $x\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle$. The available term $\mathsf{cp}(x, \alpha_0)$ is of the

$$\vdash \Delta \rightsquigarrow e[\bullet]$$

$$\frac{}{\vdash \varepsilon \rightsquigarrow \bullet} \quad \text{(abs-1)}$$

$$\frac{\vdash \Delta \rightsquigarrow e[\bullet]}{\vdash x\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: q, \Delta \rightsquigarrow \lambda \mathtt{cp}(x, \alpha_0) \rightarrow e[\bullet]} \quad \text{(abs-2)}$$

Figure 6.11: Conversion of dependency constraints into explicit abstractions, extends Figure 6.10

type corresponding to q_1 and can thus be converted to the type corresponding to q_2 by $e[\bullet]$, because $q_1 \leqslant q_2$.

We will prove the correctness of the subsumption relation on qualified types in Section 6.5.

6.4.4 Translation of expressions

The judgments for expressions are of the form

$$[\![e_{\text{FCR+tif+par}} :: t_{\text{FCR+tif+par}}]\!]^{\text{par}}_{\text{K};\Gamma;\Delta;\Sigma} \equiv e_{\text{FCR}} \quad .$$

The expression $e_{\text{FCR+tif+par}}$ is checked against an unqualified type $t_{\text{FCR+tif+par}}$, and translated to the expression e_{FCR}. As announced earlier, we make use of four environments: the kind environment K, the type environment Γ, the environment of available dependencies Δ, and the signature environment Σ.

For most syntactic constructs in the expression language, the combination of type checking and translation is entirely straightforward. As an example consider the rule for application, which is

$$\frac{\begin{array}{c}[\![e_1 :: t_1 \rightarrow t_2]\!]^{\text{par}} \equiv e_1' \\ [\![e_2 :: t_1]\!]^{\text{par}} \equiv e_2'\end{array}}{[\![(e_1\ e_2) :: t_2]\!]^{\text{par}} \equiv (e_1'\ e_2')} \quad \text{(e/tr-app)} \quad .$$

The type checking parts are analogous to (e-app), and the translation is the translation of the subexpressions. Figure 6.13 lists the cases that deal in one way or another with type-indexed functions, namely let statements (where the declarations might contain definitions of type-indexed functions), and generic application.

$$\mathrm{K}; \Delta_1 \Vdash \Delta_2 \rightsquigarrow e[\bullet]$$

$$\frac{}{\mathrm{K}; \Delta \Vdash \varepsilon \rightsquigarrow \bullet} \quad \text{(ent-1)}$$

$$\frac{\begin{array}{c}\mathrm{K}; \Delta_1 \Vdash \Delta_2 \rightsquigarrow e_\Delta[\bullet] \\ x\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: q_1 \in \Delta_1 \\ \mathrm{K}; \Delta_1 \vdash q_1 \leqslant q_2 \rightsquigarrow e[\bullet]\end{array}}{\mathrm{K}; \Delta_1 \Vdash x\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: q_2, \Delta_2 \rightsquigarrow (e_\Delta[\bullet\ e[\mathsf{cp}(x, \alpha_0)]])} \quad \text{(ent-2)}$$

Figure 6.12: Entailment of dependency constraints, extends Figure 6.10

$$[\![e_{\text{FCR+tif+par}} :: t_{\text{FCR+tif+par}}]\!]^{\text{par}}_{\mathrm{K};\Gamma;\Delta;\Sigma} \equiv e_{\text{FCR}}$$

$$\frac{\begin{array}{c}\Gamma' \equiv \Gamma\ \{, \Gamma_i\}^{i \in 1..n} \qquad \Sigma' \equiv \Sigma\ \{, \Sigma_i\}^{i \in 1..n} \\ \left\{ [\![d_i \rightsquigarrow \Gamma_i; \Sigma_i]\!]^{\text{par}}_{\mathrm{K};\Gamma';\Delta;\Sigma'} \equiv \{d_{i,j}\}^{j \in 1..m_i} \right\}^{i \in 1..n} \\ [\![e :: t]\!]^{\text{par}}_{\mathrm{K};\Gamma';\Delta;\Sigma'} \equiv e'\end{array}}{[\![\textbf{let}\ \{d_i\}_{;}^{i \in 1..n}\ \textbf{in}\ e :: t]\!]^{\text{par}}_{\mathrm{K};\Gamma;\Delta;\Sigma} \equiv \textbf{let}\ \left\{ \{d_{i,j}\}_{;}^{j \in 1..m_i} \right\}_{;}^{i \in 1..n}\ \textbf{in}\ e'} \quad \text{(e/tr-let)}$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash A :: * \\ \mathsf{gapp}_{\mathrm{K};\Gamma}(x\ \langle A \rangle) \equiv q \qquad [\![x\ \langle A \rangle]\!]^{\text{gtrans}}_{\mathrm{K};\Gamma;\Sigma} \equiv e \\ \mathrm{K}; \Delta \vdash q \leqslant t\end{array}}{[\![x\ \langle A \rangle :: t]\!]^{\text{par}}_{\mathrm{K};\Gamma;\Delta;\Sigma} \equiv e} \quad \text{(e/tr-genapp)}$$

Figure 6.13: Translation of FCR+tif+par expressions to FCR

$$[\![e_{\text{FCR+tif+par}} :: q_{\text{FCR+tif+par}}]\!]^{\text{par}}_{\text{K};\Gamma;\Delta;\Sigma} \equiv e_{\text{FCR}}$$

$$\frac{[\![e :: t]\!]^{\text{par}}_{\Delta,\Delta'} \equiv e' \qquad \Delta \vdash t \leqslant (\Delta') \Rightarrow t \rightsquigarrow e''[\bullet]}{[\![e :: (\Delta') \Rightarrow t]\!]^{\text{par}}_{\Delta} \equiv e''[e']} \quad \text{(e/tr-reveal)}$$

$$\frac{a \notin \mathsf{ftv}(\Gamma) \qquad [\![e :: q]\!]^{\text{par}}_{\Gamma} \equiv e'}{[\![e :: \forall a.q]\!]^{\text{par}}_{\Gamma} \equiv e'} \quad \text{(e/tr-gen)} \qquad \frac{[\![e :: q_1]\!]^{\text{par}} \equiv e' \qquad \vdash q_1 \leqslant q_2 \rightsquigarrow e''[\bullet]}{[\![e :: q_2]\!]^{\text{par}} \equiv e''[e']} \quad \text{(e/tr-subs)}$$

Figure 6.14: Revelation of dependency constraints in FCR+tif+par

The rule (e/tr-let) for let statements is a combination of the type checking rule (e-let) and the translation rule (tr-let), both from Chapter 4. In this rule, we make use of the correctness and translation judgment for declarations, which will be defined in Section 6.4.5.

The rule for generic application (e/tr-genapp) is different, though: the type argument is checked to be of kind $*$. The generic application algorithm from Section 6.3 is invoked to get a qualified type q for the application $x\ \langle A \rangle$. Independently, a translation algorithm is used to obtain an FCR expression e for $x\ \langle A \rangle$. The translation algorithm will be discussed later in this section. If the qualified type q has dependency constraints, the expression e is an expression that contains references to these dependency constraints. We use the judgment $\text{K}; \Delta \vdash q \leqslant t$ to check that the current dependency environment Δ contains enough information, such that in the translation, all variables that e refers to are indeed in scope at this position.

In addition to the rules just discussed, Figure 6.14 shows three rules for judgments of the form

$$[\![e_{\text{FCR+tif+par}} :: q]\!]^{\text{par}}_{\text{K};\Gamma;\Delta;\Sigma} \equiv e_{\text{FCR}}\ ,$$

where a qualified type is assigned to an expression. We will make use of these rules while checking the arms of a **typecase** statement.

Most important is rule (e/tr-reveal), which can be used to make the dependency constraints of a type, which are usually implicitly recorded in the Δ environment passed around, explicit as a qualified type.

The other two rules, (e/tr-gen) and (e/tr-subs), are counterparts of the former generalization and subsumption rules (e-gen) and (e-subs).

6.4.5 Translation of declarations

Checking and translation of declarations is shown in Figure 6.15, with judgments of the form

$$[\![d_{\text{FCR+tif+par}} \rightsquigarrow \Gamma_2; \Sigma_2]\!]^{\text{par}}_{K;\Gamma_1;\Delta;\Sigma_1} \equiv \{d_{\text{FCR}}\}^{i \in 1..n}_{;} \ ,$$

Other than in the judgments for expressions, the translation is a list of FCR declarations. As a byproduct, new bindings for the type environment, and possibly new signature entries for definitions of type-indexed functions are generated in Γ_2 and Σ_2.

The rule for declarations of type-indexed functions is the interesting one. Let us first look at the result: a well-formed type signature must exist and is returned for the type environment, and a signature environment is produced that contains one entry for each arm of the typecase. The translation consists of declarations of components, each of which also stems from one arm of the typecase.

The translation of the arms is handled by an auxiliary rule (d/tr-arm) of the form

$$[\![x \mid P \to e_{\text{FCR+tif+par}} \rightsquigarrow \Sigma_2]\!]^{\text{par}}_{K;\Gamma;\Delta;\Sigma_1} \equiv d_{\text{FCR}} \ .$$

We take the name of the function x and a single arm $P \to e_{\text{FCR+tif+par}}$ as input. Compared to the declaration judgments, we produce only one declaration and no type environment.

The type pattern P is of the form $T\ \{\alpha_i\}^{i \in 1..n}$, a named type applied to dependency variables. We implicitly assume that all type constructors T that appear in the type patterns of a typecase are distinct. The named type T is, together with the name of the function x, used to name the resulting component $\mathsf{cp}(x, T)$. The generic application algorithm gapp is used to derive the type q that the arm should have. The type q determines which dependencies may be required by expression on the right hand side. That expression is therefore checked to be of this qualified type q, making use of the rules in Figure 6.14. The translation of the expression, e', is used as the body of the component.

Compared to a type-indexed function, a normal value definition is trivial to check and translate: The type of the right hand side determines the binding for the type environment that is returned. No new signature entries are produced. The translation is a value declaration again, assigning the same name to the translation of the right hand side.

$$[\![d_{\text{FCR+tif+par}} \rightsquigarrow \Gamma_2; \Sigma_2]\!]^{\text{par}}_{\text{K};\Gamma_1;\Delta;\Sigma_1} \equiv \{d_{\text{FCR}}\}^{i \in 1..n}_{;}$$

$$\frac{\begin{array}{c} \text{K}; \Gamma \vdash^{tpsig} x\ \langle a :: *\rangle :: \sigma \\ \Gamma' \equiv x\ \langle a :: *\rangle :: \sigma \qquad \Sigma' \equiv \{\Sigma_i\}^{i\in 1..n}_{,} \\ \left\{ [\![P_i \to e_i \rightsquigarrow \Sigma_i]\!]^{\text{par}}_{\text{K};\Gamma,\Gamma';\Delta;\Sigma} \equiv d_i \right\}^{i\in 1..n} \end{array}}{[\![x\ \langle a\rangle = \textbf{typecase}\ a\ \textbf{of}\ \{P_i \to e_i\}^{i\in 1..n}_{;} \rightsquigarrow \Gamma'; \Sigma']\!]^{\text{par}}_{\text{K};\Gamma;\Delta;\Sigma} \equiv \{d_i\}^{i\in 1..n}_{;}} \quad \text{(d/tr-tif)}$$

$$\frac{[\![e :: t]\!]^{\text{par}} \equiv e'}{[\![x = e \rightsquigarrow e :: t; \varepsilon]\!]^{\text{par}} \equiv x = e'} \quad \text{(d/tr-fdecl)}$$

$$[\![x \mid P \to e_{\text{FCR+tif+par}} \rightsquigarrow \Sigma_2]\!]^{\text{par}}_{\text{K};\Gamma;\Delta;\Sigma_1} \equiv d_{\text{FCR}}$$

$$\frac{\begin{array}{c} \text{K} \vdash^{pat} P :: * \rightsquigarrow \text{K}' \qquad P \equiv T\ \{\alpha_i\}^{i\in 1..n} \\ \mathsf{gapp}_{\text{K},\text{K}';\Gamma}(x\ \langle P\rangle) \equiv q \qquad [\![e :: q]\!]^{\text{par}}_{\text{K},\text{K}';\Gamma;\Delta;\Sigma} \equiv e' \end{array}}{[\![x \mid P \to e \rightsquigarrow x\ \langle T\rangle]\!]^{\text{par}}_{\text{K};\Gamma;\Delta;\Sigma} \equiv \mathsf{cp}(x, T) = e'} \quad \text{(d/tr-arm)}$$

Figure 6.15: Translation of FCR+tif+par declarations to FCR

$$[\![K_{\text{FCR}+\text{tif}+\text{par}}]\!]^{\text{par}} \equiv K_{\text{FCR}}$$

$$\frac{}{[\![\varepsilon]\!]^{\text{par}} \equiv \varepsilon} \quad \text{(par-kap-1)} \qquad \frac{}{[\![K, a :: \kappa]\!]^{\text{par}} \equiv [\![K]\!]^{\text{par}}, a :: \kappa} \quad \text{(par-kap-2)}$$

$$\frac{}{[\![K, \alpha :: \kappa]\!]^{\text{par}} \equiv [\![K]\!]^{\text{par}}} \quad \text{(par-kap-3)}$$

Figure 6.16: Translation of FCR+tif+par kind environments to FCR

6.4.6 Translation of environments

Before we can hope to state any results about the correctness of the translation, we also need a way to map the four environments that are needed to translate FCR+tif+par programs to the two environments (kind environment and type environment) that are required to check FCR programs.

In Figure 4.5, we have mapped a type and a signature environment in language FCR+tif to a type environment FCR. We must now extend that algorithm to incorporate the dependency environment.

Figure 6.16 shows how to translate FCR+tif+par kind environments to FCR. This is achieved by removing all bindings for dependency variables, and keeping everything else.

Figure 6.17 shows how to translate type environments, signature environments, and dependency environments, all resulting in FCR type environments. To translate a type environment, we remove all type signatures for type-indexed functions, but keep the rest.

If a signature environment contains an entry $x \langle T \rangle$, this is translated into a binding for $\mathsf{cp}(x, T)$. We now use gapp to first determine the qualified type that is associated with the component, and then translate that qualified type to an FCR type. Because the gapp expects a type argument of kind $*$, we introduce fresh dependency variables as arguments of T if needed.

An entry $x \langle \alpha_0 \; \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} :: q \rangle$ of a dependency environment is translated into a binding for $\mathsf{cp}(x, \alpha_0)$, and the type that is associated with that component is the translation of q.

In the following, we use

$$[\![K; \Gamma; \Delta; \Sigma]\!]^{\text{par}}_{K;\Gamma} \equiv [\![K]\!]^{\text{par}}; [\![\Gamma]\!]^{\text{par}}, [\![\Delta]\!]^{\text{par}}, [\![\Sigma]\!]^{\text{par}}_{K;\Gamma}$$

as an abbreviation.

$$[\![\Gamma_{\text{FCR+tif+par}}]\!]^{\text{par}} \equiv \Gamma_{\text{FCR}}$$

$$\frac{}{[\![\varepsilon]\!]^{\text{par}} \equiv \varepsilon} \quad (\text{par-gam-1}) \qquad \frac{}{[\![\Gamma, x :: t]\!]^{\text{par}} \equiv [\![\Gamma]\!]^{\text{par}}, x :: t} \quad (\text{par-gam-2})$$

$$\frac{}{[\![\Gamma, x \ \langle a :: *\rangle :: \sigma]\!]^{\text{par}} \equiv [\![\Gamma]\!]^{\text{par}}} \quad (\text{par-gam-3})$$

$$[\![\Sigma_{\text{FCR+tif+par}}]\!]^{\text{par}}_{\text{K}_{\text{FCR+tif+par}};\Gamma_{\text{FCR+tif+par}}} \equiv \Gamma_{\text{FCR}}$$

$$\frac{}{[\![\varepsilon]\!]^{\text{par}} \equiv \varepsilon} \quad (\text{par-sig-1})$$

$$\frac{\begin{array}{c}\text{K} \vdash T :: \{\kappa_i \to\}^{i \in 1..n} * \\ \{\alpha_i \text{ fresh}\}^{i \in 1..n} \qquad \text{K}' \equiv \text{K}\ \{, \alpha_i :: \kappa_i\}^{i \in 1..n}\end{array}}{[\![\Sigma, x \ \langle T\rangle]\!]^{\text{par}}_{\text{K};\Gamma} \equiv [\![\Sigma]\!]^{\text{par}}_{\text{K};\Gamma}, \mathsf{cp}(x, T) :: [\![\mathsf{gapp}_{\text{K}';\Gamma}(x \ \langle T \ \{\alpha_i\}^{i \in 1..n}\rangle)]\!]^{\text{par}}} \quad (\text{par-sig-2})$$

$$[\![\Delta_{\text{FCR+tif+par}}]\!]^{\text{par}} \equiv \Gamma_{\text{FCR}}$$

$$\frac{}{[\![\varepsilon]\!]^{\text{par}} \equiv \varepsilon} \quad (\text{par-dep-1})$$

$$\frac{}{[\![\Delta, x \ \langle \alpha_0 \ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n}\rangle :: q]\!]^{\text{par}} \equiv [\![\Delta]\!]^{\text{par}}, \mathsf{cp}(x, \alpha_0) :: [\![q]\!]^{\text{par}}} \quad (\text{par-dep-2})$$

Figure 6.17: Translation of FCR+tif+par environments to FCR type environments

$$[\![x\ \langle A\rangle]\!]^{\text{gtrans}}_{K;\Gamma;\Sigma} \equiv e_{\text{FCR}}$$

$$\frac{x\ \langle T\rangle \in \Sigma}{[\![x\ \langle T\rangle]\!]^{\text{gtrans}}_{K;\Gamma;\Sigma} \equiv \mathsf{cp}(x, T)} \quad \text{(gtr-named)}$$

$$\frac{}{[\![x\ \langle \alpha\rangle]\!]^{\text{gtrans}} \equiv \mathsf{cp}(x, \alpha)} \quad \text{(gtr-depvar)}$$

$$\frac{\mathsf{dependencies}_\Gamma(x) \equiv \{y_k\}^{k\in 1..\ell}_{,}}{[\![x\ \langle A_1\ A_2\rangle]\!]^{\text{gtrans}}_{K;\Gamma;\Sigma} \equiv [\![x\ \langle A_1\rangle]\!]^{\text{gtrans}}_{K;\Gamma;\Sigma}\ \{[\![y_k\ \langle A_2\rangle]\!]^{\text{gtrans}}_{K;\Gamma;\Sigma}\}^{k\in 1..\ell}} \quad \text{(gtr-app)}$$

Figure 6.18: Translation of generic application in FCR+tif+par

6.4.7 Translation of generic application

The final topic that we need to discuss is the translation of generic application (as opposed to the algorithm gapp that determines the type of a generic application). The judgment is of the form

$$[\![x\ \langle A\rangle]\!]^{\text{gtrans}}_{K;\Gamma;\Sigma} \equiv e_{\text{FCR}}\ ,$$

and is for use in situations where the dependency constraints that are required by the translation of $x\ \langle A\rangle$ are already in scope. References to the constraints occur in e_{FCR} as free variables. The kind of the type argument A is not restricted to $*$. Although external calls will always provide a type argument A of kind $*$, the recursive calls may be on type arguments of higher kinds as well.

The algorithm is defined in Figure 6.18. We have used the algorithm in rule (e/tr-genapp) of Figure 6.13.

We have sketched the idea of the translation in Section 5.4. A generic application is translated into an application of components. If dependency variables appear in the type argument, components of the form $\mathsf{cp}(x, \alpha)$, referring to the component of a generic function for some dependency variable, are used in the translation. We call such components **dependency components**. The translation as a whole has to make sure that all components that are used – both normal and dependency components – are also available. For normal components, the signature environment Σ can be used to check which ones are available. For dependency components, the type system takes care that only available components are referred to. The environment Δ does not play a role in the $[\![\cdot]\!]^{\mathrm{gtrans}}$ algorithm.

The first two rules, (gtr-named) and (gtr-depvar) are the base cases of the algorithm. If the type argument is a named type or a dependency variable, we can translate the call directly to the appropriate component. In the case of a named type, we have to require that the component is available, and thus require an entry of the form $x\ \langle T\rangle$ to be contained in the signature environment. Note that the components $\mathsf{cp}(x, T)$ or $\mathsf{cp}(x, \alpha)$ are functions in the case that the type argument is not of kind $*$.

If none of the base cases is applicable, the type argument is an application, and the call is of the form $x\ \langle A_1\ A_2\rangle$. We recursively translate the call $x\ \langle A_1\rangle$, where A_1 is now of functional kind! The translation of this call is a function that expects arguments corresponding to the dependencies of x. We thus provide the translations of the calls $y_k\ \langle A_2\rangle$ – where the y_k are the dependencies of x – as arguments. We will show in Theorem 6.5 that the result is type correct in FCR.

6.5 Correctness of the translation

In the previous section, we have explained how the additional features that we have introduced in FCR+tif+par, such as parametrized type patterns, dependencies, and compound type arguments in calls to type-indexed functions, can be translated into the core language FCR. This is significantly more difficult than translating the plain type-indexed functions in language FCR+tif, which was the topic of Chapter 4.

We have defined a translation on expressions, declarations, (qualified) types, and environments, and it is of course desirable that the result of the translation is indeed a correct FCR program.

Theorem 6.2 (Correctness of FCR+tif+par). If we can check and translate an expression in FCR+tif+par, i.e.,

$$[\![e :: q]\!]^{\mathrm{par}}_{\mathrm{K};\Gamma;\Delta;\Sigma} \equiv e' \ ,$$

then the translation is type correct in FCR, and

$$[\![\mathrm{K};\Gamma;\Delta;\Sigma]\!]^{\mathrm{par}}_{\mathrm{K};\Gamma} \vdash e' :: [\![q]\!]^{\mathrm{par}} \ .$$

The theorem does not only state that the translation is type correct, but also that the FCR type of the translated expression is the translated qualified type of the original expression. The following corollary summarizes a few interesting special cases:

Corollary 6.3. If we can check an expression to be of an unqualified type, i.e.,

$$[\![e :: t]\!]^{\mathrm{par}}_{\mathrm{K};\Gamma;\Delta;\Sigma} \equiv e' \ ,$$

then the translation is of the same type,

$$[\![\mathrm{K};\Gamma;\Delta;\Sigma]\!]^{\mathrm{par}}_{\mathrm{K};\Gamma} \vdash e' :: t \ .$$

If Γ does not contain any type-indexed functions and K does not contain any dependency variables, we even have $\mathrm{K};\Gamma \vdash e' :: t$.

The proof of the theorem proceeds once more by induction on the combined typing and translation derivation for the original expression. Several different judgment forms are involved in such a derivation, and many of them are mutually dependent. Therefore, for each of these judgments, a variant of the correctness theorem has to formulated, and the induction has to be performed with respect to all of these judgments.

We will pick two relatively interesting parts of the derivation and state them as separate theorems and give proofs for these parts, while assuming correctness of the other parts.

Theorem 6.4 (Correctness of subsumption for qualified types). If $\mathrm{K};\Delta \vdash q_1 \leqslant q_2 \rightsquigarrow e[\bullet]$, Γ is a type environment, and e' is an FCR expression with

$$[\![\mathrm{K}]\!]^{\mathrm{par}};\Gamma,[\![\Delta]\!]^{\mathrm{par}} \vdash e' :: [\![q_1]\!]^{\mathrm{par}} \ ,$$

then

$$[\![\mathrm{K}]\!]^{\mathrm{par}};\Gamma,[\![\Delta]\!]^{\mathrm{par}} \vdash e[e'] :: [\![q_2]\!]^{\mathrm{par}} \ .$$

Proof. The subsumption relation (cf. Figure 6.10) makes significant use of two auxiliary judgments. The conversion into abstractions in Figure 6.11, and the entailment relation in Figure 6.12.

For both, we can state variants of the correctness claim. Let us start with the abstraction judgment. The statement to prove here is that if $\vdash \Delta \rightsquigarrow e[\bullet]$, Γ is a type environment, e' is an FCR expression and q a qualified type such that

$$[\![\mathrm{K}]\!]^{\mathrm{par}};\Gamma,[\![\Delta]\!]^{\mathrm{par}} \vdash e' :: [\![q]\!]^{\mathrm{par}} \ ,$$

then

$$[\![\mathrm{K}]\!]^{\mathrm{par}};\Gamma \vdash e[e'] :: [\![(\Delta) \Rightarrow q]\!]^{\mathrm{par}} \ .$$

The case (abs-1) is trivial. To prove case (abs-2), we assume that we know that

$$[\![K]\!]^{\text{par}}; \Gamma, [\![x \ \langle \alpha_0 \ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: q', \Delta]\!]^{\text{par}} \vdash e' :: [\![q]\!]^{\text{par}} \ .$$

Of course, we assume during this proof that the dependency constraints in all dependency environments appear in ascending order. We know that

$$[\![x \ \langle \alpha_0 \ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: q', \Delta]\!]^{\text{par}} \equiv \mathsf{cp}(x, \alpha_0) :: [\![q']\!]^{\text{par}}, [\![\Delta]\!]^{\text{par}} \ .$$

We can therefore apply the induction hypothesis to

$$[\![K]\!]^{\text{par}}; \Gamma, \mathsf{cp}(x, \alpha_0) :: [\![q']\!]^{\text{par}}, [\![\Delta]\!]^{\text{par}} \vdash e' :: [\![q]\!]^{\text{par}}$$

and get

$$[\![K]\!]^{\text{par}}; \Gamma, \mathsf{cp}(x, \alpha_0) :: [\![q']\!]^{\text{par}} \vdash e[e'] :: [\![(\Delta) \Rightarrow q]\!]^{\text{par}} \ .$$

Applying (e-lam) now yields

$$[\![K]\!]^{\text{par}}; \Gamma \vdash (\lambda \mathsf{cp}(x, \alpha_0) \to e[e']) :: [\![q']\!]^{\text{par}} \to [\![(\Delta) \Rightarrow q]\!]^{\text{par}} \ .$$

Observing that

$$[\![q']\!]^{\text{par}} \to [\![(\Delta) \Rightarrow q]\!]^{\text{par}} \equiv [\![(x \ \langle \alpha_0 \ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: q', \Delta) \Rightarrow q]\!]^{\text{par}} \ ,$$

we are done.

Next, we will deal with the entailment judgment. Here, the adapted correctness statement we are proving states that if $K; \Delta_1 \Vdash \Delta_2 \rightsquigarrow e_\Delta[\bullet]$, and Γ is a type environment, e' an FCR expression, and q a qualified type such that

$$[\![K]\!]^{\text{par}}; \Gamma, [\![\Delta_1]\!]^{\text{par}} \vdash e' :: [\![(\Delta_2) \Rightarrow q]\!]^{\text{par}} \ ,$$

then

$$[\![K]\!]^{\text{par}}; \Gamma, [\![\Delta_1]\!]^{\text{par}} \vdash e_\Delta[e'] :: [\![q]\!]^{\text{par}} \ .$$

Again, the case (ent-1) is trivial. For case (ent-2), we assume that there are K, Γ, Δ_1 and Δ_2 such that

$$[\![K]\!]^{\text{par}}; \Gamma, [\![\Delta_1]\!]^{\text{par}} \vdash e' :: [\![(x \ \langle \alpha_0 \ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: q_2, \Delta_2) \Rightarrow q]\!]^{\text{par}} \ .$$

We know that $x \ \langle \alpha_0 \ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: q_1$ is in Δ_1 – that is a condition on (ent-2). Therefore,

$$[\![K]\!]^{\text{par}}; \Gamma, [\![\Delta_1]\!]^{\text{par}} \vdash \mathsf{cp}(x, \alpha_0) :: [\![q_1]\!]^{\text{par}} \ .$$

To this judgment we can apply the induction hypothesis (on the subsumption statement $K; \Delta_1 \vdash q_1 \leqslant q_2 \rightsquigarrow e[\bullet]$ in the antecedent), and conclude that

$$[\![K]\!]^{\mathrm{par}}; \Gamma, [\![\Delta_1]\!]^{\mathrm{par}} \vdash e[\mathsf{cp}(x, \alpha_0)] :: [\![q_2]\!]^{\mathrm{par}} \quad .$$

Next, we apply rule (e-app), which gives us

$$[\![K]\!]^{\mathrm{par}}; \Gamma, [\![\Delta_1]\!]^{\mathrm{par}} \vdash (e'\ e[\mathsf{cp}(x, \alpha_0)]) :: [\![(\Delta_2) \Rightarrow q]\!]^{\mathrm{par}} \quad .$$

Applying the induction hypothesis for the entailment $K; \Delta_1 \Vdash \Delta_2 \rightsquigarrow e_\Delta[\bullet]$ to the above judgment results in

$$[\![K]\!]^{\mathrm{par}}; \Gamma, [\![\Delta_1]\!]^{\mathrm{par}} \vdash e_\Delta[e'\ e[\mathsf{cp}(x, \alpha_0)]] :: [\![q]\!]^{\mathrm{par}} \, ,$$

which is what we require.

We are now going to prove the rules of the subsumption judgment itself. For case (qs-dep-1), we assume that

$$[\![K]\!]^{\mathrm{par}}; \Gamma, [\![\Delta_1]\!]^{\mathrm{par}} \vdash e' :: [\![q]\!]^{\mathrm{par}} \quad .$$

Then, of course, also

$$[\![K]\!]^{\mathrm{par}}; \Gamma, [\![\Delta_1]\!]^{\mathrm{par}}, [\![\Delta_2]\!]^{\mathrm{par}} \vdash e' :: [\![q]\!]^{\mathrm{par}} \, ,$$

which under the induction hypothesis (on the subsumption $\Delta_1, \Delta_2 \vdash q \leqslant t \rightsquigarrow e[\bullet]$) yields

$$[\![K]\!]^{\mathrm{par}}; \Gamma, [\![\Delta_1]\!]^{\mathrm{par}}, [\![\Delta_2]\!]^{\mathrm{par}} \vdash e[e'] :: [\![t]\!]^{\mathrm{par}} \quad .$$

We can apply the induction hypothesis once more (on the abstraction $\vdash \Delta_2 \rightsquigarrow e_\Delta[\bullet]$) to get

$$[\![K]\!]^{\mathrm{par}}; \Gamma, [\![\Delta_1]\!]^{\mathrm{par}} \vdash e_\Delta[e[e']] :: [\![(\Delta_2) \Rightarrow t]\!]^{\mathrm{par}} \quad .$$

Next, case (qs-dep-2). We assume

$$[\![K]\!]^{\mathrm{par}}; \Gamma, [\![\Delta_2]\!]^{\mathrm{par}} \vdash e' :: [\![(\Delta_1) \Rightarrow t_1]\!]^{\mathrm{par}} \quad .$$

Using the induction hypothesis (on the entailment $K; \Delta_2 \Vdash \Delta_1 \rightsquigarrow e[\bullet]$), we can conclude

$$[\![K]\!]^{\mathrm{par}}; \Gamma, [\![\Delta_2]\!]^{\mathrm{par}} \vdash e[e'] :: [\![t_1]\!]^{\mathrm{par}} \quad .$$

Because $[\![t_1]\!]^{\mathrm{par}} \equiv t_1$ as well as $[\![t_2]\!]^{\mathrm{par}} \equiv t_2$ and $[\![K]\!]^{\mathrm{par}} \vdash t_1 \leqslant t_2$, we can apply (e-subs) and are done.

For (qs-skol), we assume that

$$[\![K]\!]^{\mathrm{par}}; \Gamma, [\![\Delta]\!]^{\mathrm{par}} \vdash e' :: [\![q_1]\!]^{\mathrm{par}}$$

and that a is a type variable that does not occur free in q_1, thus neither in $[\![q_1]\!]^{\text{par}}$. We can furthermore assume that a does not occur in K or Γ (otherwise, rename a everywhere). From the induction hypothesis, we can conclude that

$$[\![\mathrm{K}, a :: \kappa]\!]^{\text{par}}; \Gamma, [\![\Delta]\!]^{\text{par}} \vdash e[e'] :: [\![q_2]\!]^{\text{par}} \quad .$$

We can now apply (e-gen) to get

$$[\![\mathrm{K}]\!]^{\text{par}}; \Gamma, [\![\Delta]\!]^{\text{par}} \vdash e[e'] :: [\![\forall a :: \kappa.\ q_2]\!]^{\text{par}} \ ,$$

observing that $\forall a :: \kappa.\ [\![q_2]\!]^{\text{par}} \equiv [\![\forall a :: \kappa.\ q_2]\!]^{\text{par}}$, which is what we need (we can re-rename a now everywhere but in the quantified expression).

The last case is (qs-inst): here, we assume that

$$[\![\mathrm{K}]\!]^{\text{par}}; \Gamma, [\![\Delta]\!]^{\text{par}} \vdash e' :: [\![\forall a :: \kappa.\ q_2]\!]^{\text{par}} \quad .$$

Again, $[\![\forall a :: \kappa.\ q_2]\!]^{\text{par}} \equiv \forall a :: \kappa.\ [\![q_2]\!]^{\text{par}}$. Furthermore, $[\![q_1]\!]^{\text{par}}[u\ /\ a] \equiv [\![q_1[u\ /\ a]]\!]^{\text{par}}$. We can thus apply the induction hypothesis and rule (e-subs) to get

$$[\![\mathrm{K}]\!]^{\text{par}}; \Gamma, [\![\Delta]\!]^{\text{par}} \vdash e[e'] :: [\![q_2]\!]^{\text{par}} \ ,$$

which is the desired result. □

The critical part of the whole correctness proof is the one that involves the generic application algorithm. We want to show that the translation that gapp generates for a call $x\ \langle A \rangle$ is type correct in FCR, and that the type of that translation in FCR is related to the qualified type $\mathsf{gapp}(x\ \langle A \rangle)$.

The $[\![\cdot]\!]^{\text{gtrans}}$ algorithm internally uses calls $x\ \langle A \rangle$, where A has arbitrary kind. In contrast, gapp, as given in Figure 6.8, is limited to type arguments of kind $*$. To facilitate the proof, we now lift this restriction and extend gapp to type arguments of arbitrary kind. The additional rule is displayed in Figure 6.19. The prerequisites are that the type argument is free of dependency variable. A closer inspection reveals that in the original rules of the algorithm in Figure 6.8, only the base case (ga-1) contains the restriction that the type argument must be of kind $*$. The other two, for handling dependency variables, can be applied on type arguments of higher kind, but will fall back on rule (ga-4) once all dependency variables have been eliminated.

The new rule (ga-4) itself is similar to the rule (ga-3), the case where a dependency variable of kind κ occurs in the type argument A, but not in the head. Here, the type argument that A has been abstracted over plays a role similar to a dependency variable. If A is of kind $\kappa \to \kappa'$, we introduce a new type variable with kind κ. Instead of dependencies, we generate explicit function arguments for each of the generic functions y_k, which we assume to be listed in canonical

$$\mathsf{gapp}_{K;\Gamma}(x\ \langle A\rangle) \equiv q$$

$$\frac{\begin{array}{c}\mathsf{fdv}(A) \equiv \varepsilon \qquad K \vdash A :: \kappa \to \kappa' \\ a \text{ fresh} \qquad K' \equiv K, a :: \kappa \\ \mathsf{dependencies}_{\Gamma}(x) \equiv \{y_k\}_{,}^{k\in 1..n}\end{array}}{\mathsf{gapp}_{K;\Gamma}(x\ \langle A\rangle) \equiv \forall a :: \kappa.\ \{\mathsf{gapp}_{K';\Gamma}(y_k\ \langle a\rangle) \to\}^{k\in 1..n}\ \mathsf{gapp}_{K';\Gamma}(x\ \langle A\ a\rangle)} \quad \text{(ga-4)}$$

Figure 6.19: Generic application algorithm extension for type arguments of higher kinds, extends Figure 6.8

ascending order. The types of the function arguments are computed by recursive calls to $\mathsf{gapp}(y_k\ \langle a\rangle)$, and the result type is given by $\mathsf{gapp}(x\ \langle A\ a\rangle)$, where type application takes the place of the substitution of the dependency variable.

Given the extended generic application algorithm, we are now capable of stating the correctness of the translation of generic application.

Theorem 6.5 (Correctness of generic application). If $\mathsf{gapp}_{K;\Gamma}(x\ \langle A\rangle) \equiv q$ and $K; \Delta \vdash q \leqslant t$, then

$$[\![K;\Gamma;\Delta;\Sigma]\!]^{\text{par}}_{K;\Gamma} \vdash [\![x\ \langle A\rangle]\!]^{\text{gtrans}}_{K;\Gamma;\Sigma} :: t\ .$$

The theorem states that if the gapp algorithm gives us a qualified type q, and we have environments in which we can satisfy all the dependency constraints that q has to yield an unqualified type t, then we can assign this type t under the translated environments to the translated generic application. The type argument A can be of arbitrary kind – we make thus use of the generalized generic application algorithm just introduced.

Proof. We prove the theorem by induction over the $[\![\cdot]\!]^{\text{gtrans}}$ derivation in Figure 6.18.

In the case of rule (gtr-named), we know that the type argument A is a named type T. In this case, we know that $x\ \langle T\rangle$ is in Σ, and that $[\![x\ \langle T\rangle]\!]^{\text{gtrans}}_{K;\Gamma;\Sigma} \equiv \mathsf{cp}(x,T)$. Now, the translation of signature environments, in particular (par-sig-2), reveals that $[\![K;\Gamma;\Delta;\Sigma]\!]^{\text{par}}$ contains the binding

$$\mathsf{cp}(x,T) :: [\![\mathsf{gapp}_{K';\Gamma}(x\ \langle T\ \{\alpha_i\}^{i\in 1..n}\rangle)]\!]^{\text{par}}\ ,$$

where, by construction of the rule (ga-4),

$$[\![\mathsf{gapp}_{\mathrm{K}';\Gamma}(x\ \langle T\ \{\alpha_i\}^{i\in 1..n}\rangle)]\!]^{\mathrm{par}} \equiv \mathsf{gapp}_{\mathrm{K}';\Gamma}(x\ \langle T\rangle)$$

(the dependencies on the left hand side translate to the same explicit function arguments that are introduced on the right hand side by rule (ga-4)).

In case (gtr-depvar), we have $A \equiv \alpha$, and $[\![x\ \langle\alpha\rangle]\!]^{\mathrm{gtrans}} \equiv \mathsf{cp}(x,\alpha)$. By rule (ga-2) and (mkdep), we have that

$$\begin{aligned} q &\equiv \mathsf{gapp}_{\mathrm{K};\Gamma}(x\ \langle\alpha\rangle)\\ &\equiv \forall a :: \kappa\ .\Big(x\ \langle\alpha\ \{\gamma_h\}^{h\in 1..p}\rangle :: \mathsf{gapp}_{\mathrm{K}';\Gamma}\big(x\ \langle a\ \{\gamma_h\}^{h\in 1..p}\rangle\big)\Big)\\ &\qquad\qquad \Rightarrow \mathsf{gapp}_{\mathrm{K}';\Gamma}(x\ \langle a\rangle)\ , \end{aligned}$$

where κ is the kind of α, the γ_h are local dependency variables, and $\mathrm{K}' \equiv \mathrm{K}, a :: \kappa$.

We observe that for any generic function x, type argument A and type t', we have that

$$\mathsf{gapp}(x\ \langle A\rangle)[t'\ /\ a] \equiv \mathsf{gapp}(x\ \langle A[t'\ /\ a]\rangle)\ .$$

Hence, we can conclude from the precondition $\mathrm{K};\Delta \vdash q \leqslant t$ that Δ contains a binding of the form

$$x\ \langle\alpha\ \{\gamma_h\}^{h\in 1..p}\rangle :: \mathsf{gapp}_{\mathrm{K}';\Gamma}\big(x\ \langle t'\ \{\gamma_h\}^{h\in 1..p}\rangle\big)$$

for some type t' of kind κ that the type variable a has been instantiated with, and we have to show that

$$[\![\mathrm{K};\Gamma;\Delta;\Sigma]\!]^{\mathrm{par}} \vdash [\![x\ \langle\alpha\rangle]\!]^{\mathrm{gtrans}}_{\mathrm{K};\Gamma;\Sigma} :: \mathsf{gapp}_{\mathrm{K}';\Gamma}(x\ \langle t'\rangle)$$

for the same t'. (Actually, the binding in Δ may be to any type convertible into the above type, but that does not make an essential difference).

Because of the binding in Δ, the translated environments $[\![\mathrm{K};\Gamma;\Delta;\Sigma]\!]^{\mathrm{par}}$ contains a binding of the form

$$\mathsf{cp}(x,\alpha) :: [\![\mathsf{gapp}_{\mathrm{K}';\Gamma}(x\ \langle a\ \{\gamma_h\}^{h\in 1..p}\rangle)]\!]^{\mathrm{par}}\ ,$$

of which the right hand side by construction is the same as

$$\mathsf{gapp}_{\mathrm{K}';\Gamma}(x\ \langle a\rangle)\ .$$

An application of (e-var) thus shows that $\mathsf{cp}(x,\alpha)$ is of the required type.

This leaves case (gtr-app): here, $A \equiv A_1\ A_2$. To simplify the notation slightly, we assume that x has just one dependency, which we name y. It is straightforward to generalize to multiple dependencies. We can write

$$q \equiv \mathsf{gapp}_{\mathrm{K};\Gamma}(x\ \langle A_1\ A_2\rangle) \equiv \{\forall a_i :: \kappa_i.\}^{i\in 1..n}\ (\Delta') \Rightarrow \mathsf{gapp}_{\mathrm{K}';\Gamma}(x\ \langle A_1'\ A_2'\rangle)\ ,$$

where

$$\begin{aligned}\mathsf{fdv}(A_1\ A_2) &\equiv \{\alpha_i\}_{,}^{i\in 1..n}\\ A_1' &\equiv A_1\{[a_i\ /\ \alpha_i]\}^{i\in 1..n}\\ A_2' &\equiv A_2\{[a_i\ /\ \alpha_i]\}^{i\in 1..n}\ .\end{aligned}$$

The translation $[\![x\ \langle A_1\ A_2\rangle]\!]^{\text{gtrans}}$ is, according to rule (gtr-app), the application $([\![x\ \langle A_1\rangle]\!]^{\text{gtrans}}\ [\![y\ \langle A_2\rangle]\!]^{\text{gtrans}})$.

Let us first inspect the call $x\ \langle A_1\rangle$ more closely. Let $\vdash A_1 :: \kappa \to \kappa'$. We have that under K and Δ,

$$\begin{aligned}\mathsf{gapp}_{\mathrm{K};\Gamma}(x\ \langle A_1\rangle) \leqslant{}& \{\forall a_i :: \kappa_i.\}^{i\in 1..n}\ (\Delta') \Rightarrow\\ &\qquad \forall a :: \kappa.\ \mathsf{gapp}_{\mathrm{K}';\Gamma}(y\ \langle a\rangle) \to \mathsf{gapp}_{\mathrm{K}';\Gamma}(x\ \langle A_1'\ a\rangle)\\ \equiv{}& q_1\ .\end{aligned}$$

We know that

$$\mathrm{K};\Delta \vdash q \leqslant t\ ,$$

which implies

$$\mathrm{K};\Delta \vdash q_1 \leqslant \mathsf{gapp}_{\mathrm{K}';\Gamma}(y\ \langle A_2'\rangle) \to t$$

if we substitute a by A_2'. We apply the induction hypothesis to the call $x\ \langle A_1\rangle$ and the unqualified type $\mathsf{gapp}_{\mathrm{K}';\Gamma}(y\ \langle A_2'\rangle) \to t$, which yields

$$[\![\mathrm{K};\Gamma;\Delta;\Sigma]\!]^{\text{par}} \vdash [\![x\ \langle A_1\rangle]\!]^{\text{gtrans}}_{\mathrm{K};\Gamma;\Sigma} :: \mathsf{gapp}_{\mathrm{K}';\Gamma}(y\ \langle A_2'\rangle) \to t\ .$$

Now, let us inspect the call $y\ \langle A_2\rangle$. Under K and Δ,

$$\begin{aligned}\mathsf{gapp}_{\mathrm{K};\Gamma}(y\ \langle A_2\rangle) \leqslant{}& \{\forall a_i :: \kappa_i.\}^{i\in 1..n}\ (\Delta') \Rightarrow \mathsf{gapp}_{\mathrm{K}';\Gamma}(y\ \langle A_2'\rangle)\\ \equiv{}& q_2\ .\end{aligned}$$

This time, the precondition $\vdash q \leqslant t$ implies that

$$\mathrm{K};\Delta \vdash q_1 \leqslant \mathsf{gapp}_{\mathrm{K}';\Gamma}(y\ \langle A_2'\rangle)\ .$$

The induction hypothesis applied to the call $y\ \langle A_2\rangle$ and the type $\mathsf{gapp}_{\mathrm{K}';\Gamma}(y\ \langle A_2'\rangle)$ yields

$$[\![\mathrm{K};\Gamma;\Delta;\Sigma]\!]^{\text{par}} \vdash [\![y\ \langle A_2\rangle]\!]^{\text{gtrans}}_{\mathrm{K};\Gamma;\Sigma} :: \mathsf{gapp}_{\mathrm{K}';\Gamma}(y\ \langle A_2'\rangle)\ .$$

The result of $[\![x\ \langle A_1\ A_2\rangle]\!]^{\text{gtrans}}$ is the application $([\![x\ \langle A_1\rangle]\!]^{\text{gtrans}}\ [\![y\ \langle A_2\rangle]\!]^{\text{gtrans}})$. We can therefore apply rule (e-app) to get

$$[\![\mathrm{K};\Gamma;\Delta;\Sigma]\!]^{\text{par}} \vdash [\![x\ \langle A_1\ A_2\rangle]\!]^{\text{gtrans}}_{\mathrm{K};\Gamma;\Sigma} :: t\ ,$$

which proves the final case of the induction and therefore the theorem. □

6.6 Kind-indexed types?

Ralf Hinze has introduced two styles of generic function definitions, which he named after the conferences where he first presented papers on them, MPC-style (Hinze 2000*c*) and POPL-style (Hinze 2000*b*). Generic Haskell has originally been based on MPC-style definitions of type-indexed functions.

In his MPC paper, Hinze revealed that a type-indexed value possesses a *kind-indexed type*. Where a type-indexed function is defined over the structure of types, a kind-indexed type is defined over the (much simpler) structure of kinds. A function such as *add* would make use of the following kind-indexed type:

$$\begin{array}{ll} \textbf{type}\ \mathrm{Add}\ \langle\!\langle * \rangle\!\rangle & a = a \to a \to a \\ \textbf{type}\ \mathrm{Add}\ \langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ a = \forall b :: \kappa.\ \mathrm{Add}\ \langle\!\langle \kappa \rangle\!\rangle\ b \to \mathrm{Add}\ \langle\!\langle \kappa' \rangle\!\rangle\ (a\ b) & \\ \mathit{add}\ \langle a :: \kappa \rangle & :: \mathrm{Add}\ \langle\!\langle \kappa \rangle\!\rangle\ a \end{array}$$

The first two lines declare a kind-indexed type – the **type** keyword indicates that we define a type-level entity, and we use double angle brackets $\langle\!\langle\cdot\rangle\!\rangle$ to denote the kind index. There are two arms, one for the kind $*$, and one for functional kinds of the form $\kappa \to \kappa'$. The argument a is supposed to be of the kind that is associated with the particular branch, so that we could assign the kind

$$\mathrm{Add}\ \langle \kappa \rangle :: \kappa \to *$$

to the kind-indexed type Add. The final line assigns the kind-indexed type to the function *add*.

For comparison, this is the type signature that we use for *add*:

$$\mathit{add}\ \langle a :: * \rangle :: (\mathit{add}) \Rightarrow a \to a \to a\ \ .$$

The Dependency-style that we use is essentially a generalization of MPC-style. For kind $*$ type arguments without dependency variables, both signatures match: if A is a type argument of kind $*$ with $\mathsf{fdv}(A) \equiv \varepsilon$, the kind-indexed type is used at Add $\langle\!\langle * \rangle\!\rangle$ A, which expands to $A \to A \to A$. Similarly $\mathsf{gapp}(x\ \langle A \rangle) \equiv \mathsf{base}(x\ \langle A \rangle) \equiv A \to A \to A$ in this case.

Differences occur if type constructors of higher kind are involved in the type arguments. In MPC-style, there are no dependency variables, nor are there dependency constraints. But neither are type arguments restricted to kind $*$ in MPC-style. Using dependency variables, we defined *add* for lists as follows:

```
add ⟨[α]⟩        x y
    | length x == length y = map (uncurry (add ⟨α⟩)) (zip x y)
    | otherwise            = error "args must have same length" .
```

In MPC-style, we would just define the arm in terms of the list type constructor $[\,]$:

$$\begin{aligned}
&\mathit{add}\ \langle[\,]\rangle\ \mathit{add}_\alpha\ x\ y\\
&\quad \mid \mathit{length}\ x \mathrel{==} \mathit{length}\ y = \mathit{map}\ (\mathit{uncurry}\ (\mathit{add}_\alpha))\quad (\mathit{zip}\ x\ y)\\
&\quad \mid \mathit{otherwise} \qquad\qquad\ \ = \mathit{error}\ \texttt{"args must have same length"}\ .
\end{aligned}$$

Instead of the explicit recursive call to *add* $\langle\alpha\rangle$, the function is provided with an extra argument add_α, that can be used in the places where we would like to call *add* on the list element type. Interestingly though, the second version of the function is (modulo renaming) identical to the component $\mathsf{cp}(\mathit{add}, [\,])$ that we translate the arm *add* $\langle[\alpha]\rangle$ to. The type of *add* $\langle[\,]\rangle$ is justified by the kind-indexed type, which states that

$$\mathrm{Add}\ \langle * \to *\rangle\ ([\,]) \equiv \forall a :: *.\ (a \to a \to a) \to [a] \to [a] \to [a]\ .$$

This corresponds to the fact that we have generalized the gapp algorithm to yield

$$\mathsf{gapp}(x\ \langle[\,]\rangle) \qquad \equiv \forall a :: *.\ (a \to a \to a) \to [a] \to [a] \to [a]\ .$$

Indeed, we use kind-indexed types internally – during the translation – because they form an expressive, theoretically solid and elegant way to treat type-indexed functions. However, on the surface, visible to the programmer and user of type-indexed functions, we make use of dependency variables instead.

The advantage of the use of dependency types is that it is more explicit: in the definition of *add* $\langle[\alpha]\rangle$, we can immediately see that *add* is recursively defined and depends on itself, whereas we have to know how the specialization mechanism works to understand that add_α – which is only the name of a bound variable here, and could have been given a much less informative name – will indeed be instantiated with the *add* function on the element type of the list in the definition of *add* $\langle[\,]\rangle$.

This becomes even clearer in the presence of type-indexed functions that have more than one dependency. Recall the type of *cpr* from Section 5.3:

$$\mathit{cpr}\ \langle a :: *\rangle :: (\mathit{equal}, \mathit{cpr}) \Rightarrow a \to a \to \mathrm{CprResult}\ .$$

The type signature of *cpr* in MPC-style would look as follows:

$$\begin{aligned}
&\textbf{type}\ \mathrm{Cpr}\ \ \ \langle\!\langle *\rangle\!\rangle \qquad\ \ a = a \to a \to \mathrm{CprResult}\\
&\textbf{type}\ \mathrm{Cpr}\ \ \ \langle\!\langle \kappa \to \kappa'\rangle\!\rangle\ a = \forall b :: \kappa.\ \mathrm{Equal}\ \langle\!\langle \kappa\rangle\!\rangle\ b \to \mathrm{Cpr}\ \langle\!\langle \kappa\rangle\!\rangle\ b\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \mathrm{Cpr}\ \langle\!\langle \kappa'\rangle\!\rangle\ (a\ b)\\
&\textbf{type}\ \mathrm{Equal}\ \langle\!\langle *\rangle\!\rangle \qquad\ \ a = a \to a \to \mathrm{Bool}\\
&\textbf{type}\ \mathrm{Equal}\ \langle\!\langle \kappa \to \kappa'\rangle\!\rangle\ a = \forall b :: \kappa.\ \mathrm{Equal}\ \langle\!\langle \kappa\rangle\!\rangle\ b \to \mathrm{Equal}\ \langle\!\langle \kappa'\rangle\!\rangle\ (a\ b)\\
&\mathit{cpr}\ \langle a :: \kappa\rangle \qquad\qquad\quad :: \mathrm{Cpr}\ \langle\!\langle \kappa\rangle\!\rangle\ a\ .
\end{aligned}$$

The kind-indexed type Cpr makes use of the kind-indexed type Equal which belongs to the *equal* function. An arm of the *cpr* function for a type constructor of higher kind is supplied with additional arguments not only for the recursive calls of *cpr*, but also for calls to *equal*.

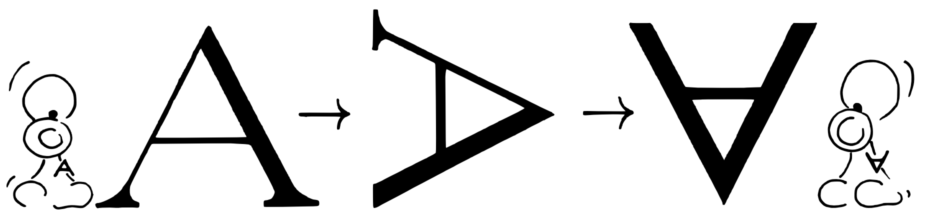
It is the programmer's responsibility to remember the order of the dependent functions and to assign suitable names to the additional arguments in order to visualize what the arguments represent. Furthermore, there needs to be some mechanism in the language that actually finds out that the function *cpr* depends on the functions *equal* and *cpr*. The names Cpr and Equal are names on the type level which we have chosen as appropriate, but Equal could be the type of several type-indexed functions, not only of *equal*, and there is no easy way for the compiler to determine the correct dependencies between the functions only from the type. Finally, if this problem has been solved, the definition is still not very robust with respect to change. Once we add an arm to a generic function that requires an additional dependency, all other arms for type constructors of higher kind suddenly get new arguments, and all definitions have to be adapted accordingly. It may be for these reasons that Hinze considers only type-indexed functions that depend exclusively on themselves in his MPC paper.

Using dependency variables, we thus achieve that we can easily determine the dependencies, because we use the names of the functions in the definitions. The dependencies are also recorded in the type. An additional dependency can be added by changing the type signature of the function – none of the arms need to be changed. Because dependencies are no longer explicit function arguments, but implicit ones, their order is irrelevant to the user.

The translation for dependencies that we have outlined in this chapter reveals that dependencies form a layer around kind-indexed types, that shields the user from its complexities and provide the user with a more explicit syntax and additional flexibility due to the practical use of functions that have multiple dependencies.

Interestingly, the syntax for type-indexed functions that we use in this thesis is very similar to the other style that Hinze introduced, POPL-style. In that approach, type constructors in type arguments are augmented with type variables, and recursive calls to type-indexed functions are explicit. Unfortunately, POPL-style is less expressive and more restricted in some respects. The Dependency-style used in this thesis, by being in essence a wrapper around MPC-style, has all the expressiveness of that style. It combines thus the advantages of both approaches: the ease of use of POPL-style with the generality of MPC-style.

7 Going Generic



With the additions of Chapter 6, we have extended the type-indexed definitions of Chapter 4 significantly, but there is still no genericity: the set of type arguments that a type-indexed function can be specialized to is exactly the set of types that can be built from the named types that are in the signature of the function.

The idea of genericity, however, is that we can write type-indexed algorithms based on the *structure* of datatypes, surpassing the differences that the names of datatypes make.

As it turns out, the remaining step, from a programmer's point of view, is a very small one. The programmer must only write a few additional arms of a type-indexed definition, for a number of special purpose datatypes, and suddenly, the function can be specialized to nearly all Haskell datatypes!

Later in this chapter, we discuss several examples of generic functions. All of them have been mentioned before and are more or less "classics" in the world of generic functions. Presenting them in the framework of Generic Haskell should give a good intuition about the way generic functions are defined and used in Generic Haskell.

7.1 Unit, Sum, and Prod

Let Unit, Sum and Prod be datatypes defined as follows:

$$\begin{aligned}&\textbf{data}\ \mathrm{Unit} &&= \mathit{Unit}\\ &\textbf{data}\ \mathrm{Sum}\ (a :: *)\ (b :: *) &&= \mathit{Inl}\ a \mid \mathit{Inr}\ b\\ &\textbf{data}\ \mathrm{Prod}\ (a :: *)\ (b :: *) &&= a \times b\ .\end{aligned}$$

These types are very basic, and in fact have their isomorphic counterparts in Haskell: Unit is isomorphic to the unit type $()$, a type with just one value (with the exception of the undefined value $\perp$, of course); Sum is isomorphic to Either, which gives a choice between two alternatives. The constructors are *Inl* (read "in-left") and *Inr* ("in-right"). Finally Prod is the same as a pair, but we use $\times$ as an infix constructor, instead of Haskell's standard pair constructor $(,)$.

These three types play a special role in the definition and use of generic functions, which is also the reason why we choose to introduce our own new types, instead of reusing the standard Haskell types. This way, $()$, Either, and $(,)$ can still be used as any other type.

Recall the type-indexed function *add*. In Chapter 4, we had defined it for Bool, Int, and Char:

$$\begin{aligned}&\mathit{add}\ \langle \mathrm{Bool}\rangle && && &&= (\vee)\\ &\mathit{add}\ \langle \mathrm{Int}\rangle && && &&= (+)\\ &\mathit{add}\ \langle \mathrm{Char}\rangle && x && y &&= \mathit{chr}\ (\mathit{ord}\ x + \mathit{ord}\ y)\ .\end{aligned}$$

It is easy to add an extra arm for Unit: there is only one value of that type, so we decide that the sum of two *Unit*'s should be *Unit* again:

$$\mathit{add}\ \langle \mathrm{Unit}\rangle\quad \mathit{Unit}\quad \mathit{Unit}\quad = \mathit{Unit}\ .$$

Later, in Chapter 6 we defined additional cases for lists and pairs. Just as for Haskell pairs, we can of course also define addition for products of type Prod, adding the two components pointwise:

$$\mathit{add}\ \langle \mathrm{Prod}\ \alpha\ \beta\rangle\ (x_1 \times x_2)\ (y_1 \times y_2) = (\mathit{add}\ \langle\alpha\rangle\ x_1\ y_1) \times (\mathit{add}\ \langle\beta\rangle\ x_2\ y_2)\ .$$

Of the three types just introduced, only a case for Sum remains to be defined. Here we take an approach similar to what we did when we tried to define addition of lists: we defined addition pointwise on the elements, provided the lists were of the same shape, i.e., length. Two values of the Sum type are of the same shape if they belong to the same alternative.

$$
\begin{array}{lllll}
\mathit{add}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle & (\mathit{Inl}\ x) & (\mathit{Inl}\ y) & = \mathit{Inl}\ (\mathit{add}\ \langle\alpha\rangle\ x\ y) \\
\mathit{add}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle & (\mathit{Inr}\ x) & (\mathit{Inr}\ y) & = \mathit{Inr}\ (\mathit{add}\ \langle\beta\rangle\ x\ y) \\
\mathit{add}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle & \_ & \_ & = \\
 & & \multicolumn{2}{l}{\quad \mathit{error}\ \texttt{"args must have same shape"}\ .}
\end{array}
$$

With the arms for Bool, Int, Char, Unit, Prod, and Sum, the function *add* becomes **generic**: the compiler can interpret applications of *add* to more datatypes than it has been defined for. How this is achieved will be the topic of Section 7.5 and later chapters. Let us first look at examples.

Using the extended definition of *add*, we can call the function on a broad variety of datatypes, for instance on lists:

$$\mathit{add}\ \langle[\mathrm{Bool}]\rangle\ [\mathit{False},\mathit{True},\mathit{True}]\ [\mathit{False},\mathit{False},\mathit{True}]$$

evaluates successfully to $[\mathit{False},\mathit{True},\mathit{True}]$, and the call

$$\mathit{add}\ \langle[\mathrm{Int}]\rangle\ [2,3]\ [1]$$

results in the error message `"args must have same shape"`. A specific definition for lists is no longer needed, because it can be derived from the cases for Unit, Sum, and Prod that have been specified. The function *add* works for trees as well: the call

$$
\begin{array}{ll}
\mathit{add}\ \langle\mathrm{Tree\ Int}\rangle & (\mathit{Node}\ (\mathit{Node\ Leaf}\ 1\ \mathit{Leaf})\ 2\ (\mathit{Node\ Leaf}\ 3\ \mathit{Leaf})) \\
 & (\mathit{Node}\ (\mathit{Node\ Leaf}\ 2\ \mathit{Leaf})\ 3\ (\mathit{Node\ Leaf}\ 5\ \mathit{Leaf}))
\end{array}
$$

evaluates to $\mathit{Node}\ (\mathit{Node\ Leaf}\ 3\ \mathit{Leaf})\ 5\ (\mathit{Node\ Leaf}\ 8\ \mathit{Leaf})$. Arbitrary combinations of types are possible, such as triples of integer lists, booleans and characters: the expression

$$\mathit{add}\ \langle([\mathrm{Int}],\mathrm{Bool},\mathrm{Char})\rangle\ ([4,2],\mathit{False},\texttt{'!'})\ ([1,3],\mathit{False},\texttt{'Y'})$$

results in $([5,5],\mathit{False},\texttt{'z'})$.

Datatypes with higher-kinded type arguments and nested datatypes may appear in the type arguments as well. The only condition is that if the type argument contains any **abstract types**, i.e., types for which no datatype definition is known, these abstract types must occur in the signature of the generic function. Examples of such abstract types are built-in types such as Int, Char, Float, or the function type constructor $(\rightarrow)$. Whereas Int and Char are part of the signature of *add*, the types Float and $(\rightarrow)$ are not. Hence, a call to

$$\mathit{add}\ \langle\mathrm{Float}\rangle$$

or also

$$\mathit{add}\ \big\langle [\,[(\mathrm{Int},\mathrm{Float})]\,] \big\rangle$$

still – notwithstanding genericity – fails at compile time with a specialization error.

In the next sections, we will – now that we finally can – discuss a few more example generic functions.

7.2 Generic enumeration

A very simple generic function computes a default value (that should somehow represent null or empty) for each datatype.

$$\begin{aligned}
&\mathit{empty}\ \langle a :: *\rangle && :: (\mathit{empty}) \Rightarrow a\\
&\mathit{empty}\ \langle \mathrm{Int}\rangle && = 0\\
&\mathit{empty}\ \langle \mathrm{Char}\rangle && = \texttt{' '}\\
&\mathit{empty}\ \langle \mathrm{Float}\rangle && = 0.0\\
&\mathit{empty}\ \langle \mathrm{Unit}\rangle && = \mathit{Unit}\\
&\mathit{empty}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle && = \mathit{Inl}\ (\mathit{empty}\ \langle\alpha\rangle)\\
&\mathit{empty}\ \langle \mathrm{Prod}\ \alpha\ \beta\rangle && = \mathit{empty}\ \langle\alpha\rangle \times \mathit{empty}\ \langle\beta\rangle\\
&\mathit{empty}\ \langle \alpha \to \beta\rangle && = \mathit{const}\ (\mathit{empty}\ \langle\beta\rangle)
\end{aligned}$$

The function is defined on the abstract types Int, Char, Float to return an appropriate value. For the Unit type, there is not much choice but to return the Unit value. For a Sum, we decide to prefer the left alternative. In a Prod, we return a pair of two empty values. Finally, we have also defined an arm for functions. Functions are another example of an abstract type: if there is no explicit arm, function types could not be handled generically. Here, we decide to return the constant function returning the empty value of the result type.

Therefore, the call

$$\mathit{map}\ \big(\lambda(x,y,z) \to (x\ \texttt{'A'},y,z)\big)\ \big(\mathit{empty}\ \langle(\mathrm{Char} \to [\mathrm{Int}],\mathrm{Tree}\ \mathrm{Float},\mathrm{Bool})\rangle\big)$$

can successfully be translated, and evaluates to $([\,],\mathit{Leaf},\mathit{False})$. The signature of *empty* is $\mathrm{Int},\mathrm{Char},\mathrm{Float},\mathrm{Unit},\mathrm{Sum},\mathrm{Prod},(\to)$, thus Bool is not contained in the signature. Nevertheless, the call above succeeds. This indicates that Bool is not considered abstract. Instead, we assume that Bool is defined via the following datatype declaration:

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

The generic function (or generic value, in this case) *empty* also serves as an example that generic functions can be used to produce, not only to consume, values of generic type. In general, it is easy to generate values of a large class of datatypes automatically using a generic function. A variant of *empty* is the function *enum* that generates *all* values of a datatype:

$$\begin{aligned}
&\mathit{enum}\ \langle a :: *\rangle && :: (\mathit{enum}) \Rightarrow [a] \\
&\mathit{enum}\ \langle \mathrm{Int}\rangle && = \mathit{interleave}\ [0, -1\,..]\ [1\,..] \\
&\mathit{enum}\ \langle \mathrm{Char}\rangle && = [\texttt{'\textbackslash NUL'}\,..] \\
&\mathit{enum}\ \langle \mathrm{Unit}\rangle && = [\mathit{Unit}] \\
&\mathit{enum}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle && = \mathit{interleave}\ (\mathit{map}\ \mathit{Inl}\ (\mathit{enum}\ \langle\alpha\rangle)) \\
& && \qquad\qquad\qquad (\mathit{map}\ \mathit{Inr}\ (\mathit{enum}\ \langle\beta\rangle)) \\
&\mathit{enum}\ \langle \mathrm{Prod}\ \alpha\ \beta\rangle && = \mathit{diag}\ (\mathit{enum}\ \langle\alpha\rangle)\ (\mathit{enum}\ \langle\beta\rangle)\ .
\end{aligned}$$

We take some precautions to handle infinite types in a way that every value will be enumerated with a finite index. The functions *interleave* and *diag* can be defined as follows:

$$\begin{aligned}
&\mathit{interleave} && :: \forall a :: *.\ [a] \to [a] \to [a] \\
&\mathit{interleave}\ [\,]\quad [\,] && = [\,] \\
&\mathit{interleave}\ [\,]\quad \mathit{ys} && = \mathit{ys} \\
&\mathit{interleave}\ (x : \mathit{xs})\ \mathit{ys} && = x : \mathit{interleave}\ \mathit{ys}\ \mathit{xs} \\
&\mathit{diag} && :: \forall (a :: *)\ (b :: *).\ [a] \to [b] \to [\mathrm{Prod}\ a\ b] \\
&\mathit{diag}\ \ \mathit{xs}\quad \mathit{ys} && = \mathit{concat}\ (\mathit{diag}'\ \mathit{xs}\ \mathit{ys}) \\
&\mathit{diag}' && :: \forall (a :: *)\ (b :: *).\ [a] \to [b] \to [[\mathrm{Prod}\ a\ b]] \\
&\mathit{diag}'\ [\,]\quad \mathit{ys} && = \mathit{map}\ (\mathit{const}\ [\,])\ \mathit{ys} \\
&\mathit{diag}'\ \mathit{xs}\quad [\,] && = \mathit{map}\ (\mathit{const}\ [\,])\ \mathit{xs} \\
&\mathit{diag}'\ (x : \mathit{xs})\ (y : \mathit{ys}) && = [(x \times y)] : \mathit{zipWith}\ (+\!\!+)\ (\mathit{diag}'\ [x]\ \mathit{ys}) \\
& && \qquad\qquad\qquad\qquad\qquad\quad (\mathit{diag}'\ \mathit{xs}\ \ (y : \mathit{ys}))\ .
\end{aligned}$$

The function *enum*, maybe with some refinements to generate a better distribution, has applications in automatic test case generation, for tools such as QuickCheck (Claessen and Hughes 2000) or GAST (Koopman *et al.* 2003). These tools allow the programmer to verify properties of their programs on a randomly generated set of example values. Because the values involved can be from a multitude of different types, the use of generic functions to generate suitable values is desirable.

Other, similar functions, include a generic function to generate the "minimal" or "maximal" value of a datatype, and a function that, given a "lower" and an "upper" bound, enumerates all values that lie in between.

7.3 Generic equality

Next, we look at the prototypical example of a generic function. An equality test is probably the most often needed operation on values of a particular datatype. Haskell has an overloaded equality function, and provides a built-in **deriving** mechanism that – on demand – automatically generates the equality function for a user-defined datatype. It seems a safe bet to guess that the reason that **deriving** is present in Haskell, is mainly the equality function. In Standard ML (Milner *et al.* 1997) equality is also treated specially, and can be automatically generated for a large class of types.

With Generic Haskell, we can now *define* the equality function.

$$\begin{array}{llll}
\mathit{equal}\ \langle \mathrm{Int} \rangle & & & = (\equiv) \\
\mathit{equal}\ \langle \mathrm{Char} \rangle & & & = (\equiv) \\
\mathit{equal}\ \langle \mathrm{Float} \rangle & & & = (\equiv) \\
\mathit{equal}\ \langle \mathrm{Unit} \rangle & \mathit{Unit} & \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inl}\ x) & (\mathit{Inl}\ y) & = \mathit{equal}\ \langle \alpha \rangle\ x\ y \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inr}\ x) & (\mathit{Inr}\ y) & = \mathit{equal}\ \langle \beta \rangle\ x\ y \\
\mathit{equal}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & \_ & \_ & = \mathit{False} \\
\mathit{equal}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle & (x_1 \times x_2) & (y_1 \times y_2) & = \mathit{equal}\ \langle \alpha \rangle\ x_1\ x_2 \wedge \mathit{equal}\ \langle \beta \rangle\ y_1\ y_2
\end{array}$$

For the primitive types Int, Char, and Float, we use Haskell's equality function $(\equiv)$. We could also use primitive equality functions on these types, but Haskell gives us access to them only via the overloaded $(\equiv)$ function.

Two Unit values are easy to compare. In the situation that we are given two sums, we distinguish three cases: both values are from the left alternative, both are from the right alternative, or they are from different alternatives. In the last case, they cannot be equal. In the other two cases, we compare the two values embedded in the sum. Products are tested for equality pointwise.

Functions are hard to compare in general, but if their domain is finite, it is possible, using *enum*:

$$\begin{array}{llll}
\mathit{equal}\ \langle \alpha \to \beta \rangle & \mathit{fx} & \mathit{fy} & = \mathit{equal}\ \langle [\beta] \rangle\ \big(\mathit{map}\ \mathit{fx}\ (\mathit{enum}\ \langle \alpha \rangle)\big) \\
 & & & \qquad\qquad\quad \big(\mathit{map}\ \mathit{fy}\ (\mathit{enum}\ \langle \alpha \rangle)\big)\ .
\end{array}$$

This function will correctly compare functions with a finite domain, and it will sooner or later return *False* for functions with an infinite domain that are not equal. If two equal functions of infinite domain are compared, the call will not terminate. (Using the generic cardinality test that is defined in Section 17.1.3, we could add a run-time check if the domain type of the function is infinite, and return a run-time error instead of a nonterminating computation.) The equality function, such as written here, depends on both itself and *enum*. Its type signature is thus

$$equal\ \langle a :: *\rangle :: (equal, enum) \Rightarrow a \to a \to \mathrm{Bool}\ .$$

Although *empty*, *enum*, and *equal* are all defined only for abstract types and the three special types Unit, Sum, and Prod, it is by no means forbidden to specify explicit behaviour also for non-abstract types. If a type is part of the signature of a generic function, the explicitly specified functionality always takes precedence over the generic behaviour.

For instance, a large class of datatypes might be used to represent an abstract syntax tree for some programming language. The abstract syntax tree is created by parsing a concrete program. To produce good error messages, it might be useful to store position information in the datatypes. All of these datatypes might have a field of type

$$\textbf{data}\ \mathrm{Range} = \mathit{Range}\ \mathrm{Position}\ \mathrm{Position}\ ,$$

storing a start and an end position, where

$$\textbf{data}\ \mathrm{Position} = \mathit{Position}\ \mathrm{String}\ \mathrm{Int}\ \mathrm{Int}\ ,$$

consisting of a filename, a line, and a column number. When checking two syntactic constructs for equality, the positional information is irrelevant. Therefore, we would like to treat two values of type Range always as equal. In this case, we simply add an additional line to the definition of *equal*:

$$equal\ \langle \mathrm{Range}\rangle\quad x\qquad y\qquad = \mathit{True}\ ,$$

and values of type Range will not be tested generically.

A generalized variant of generic equality is a generic total comparison function, that returns for any two values of a type one of the results *LT* (“less than”), *EQ* (“equal”), or *GT* (“greater than”).

7.4 Generic compression

A classic application area of generic functions is parsing and unparsing, i.e., reading values of different types from some universal representation, or writing values to that universal representation. The universal representation can be aimed at being human-readable (such as the result of Haskell’s *show* function; see also Section 17.1.2), or it can be intended for data exchange, such as XML (W3C 2004) or ATerms (Van den Brand *et al.* 2000). Other applications include encryption, transformation, or storage.

We will treat a very simple case of compression here, by defining functions that can write to and read from a sequence of bits. A bit is defined by the following datatype declaration:

$$\textbf{data}\ \mathrm{Bit} = \mathtt{0} \mid \mathtt{1}$$

(the names 0 and 1 are used as constructors here). We can now define a simple encoding function that produces a single bit whenever a choice is made:

$$\begin{array}{lll}
\mathit{encode}\ \langle a :: *\rangle & & :: (\mathit{encode}) \Rightarrow a \to [\mathrm{Bit}] \\
\mathit{encode}\ \langle \mathrm{Unit}\rangle & \mathit{Unit} & = [\,] \\
\mathit{encode}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle & (\mathit{Inl}\ x) & = \mathtt{0} : \mathit{encode}\ \langle\alpha\rangle\ x \\
\mathit{encode}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle & (\mathit{Inr}\ x) & = \mathtt{1} : \mathit{encode}\ \langle\beta\rangle\ y \\
\mathit{encode}\ \langle \mathrm{Prod}\ \alpha\ \beta\rangle & (x_1 \times x_2) & = \mathit{encode}\ \langle\alpha\rangle\ x_1 +\!\!+ \mathit{encode}\ \langle\beta\rangle\ x_2\ .
\end{array}$$

The only place where there really is a choice is in the Sum type. Here, the value can be either an *Inl* or an *Inr*. If we have to encode a value of type Unit, it can only be *Unit*, so we need no bits to encode that knowledge. Similarly, for a product we know that the value is the first component followed by the second – we need no extra bits except the encodings of the components.

The function is only really useful if we add some definitions for primitive types that encode integers or characters as suitable sequences of bits. We omit these cases here.

To recover a value from a list of bits, we define the function *decodes* (a function *decode* will be defined later, in Section 12.1):

$$\begin{array}{lll}
\mathit{decodes}\ \langle a :: *\rangle & & :: (\mathit{decodes}) \Rightarrow [\mathrm{Bit}] \to [(a,[\mathrm{Bit}])] \\
\mathit{decodes}\ \langle \mathrm{Unit}\rangle & x & = [(\mathit{Unit}\quad, x\ )] \\
\mathit{decodes}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle & (\mathtt{0} : x) & = [(\mathit{Inl}\ y\quad, r\ ) \mid (y\ , r\ ) \leftarrow \mathit{decodes}\ \langle\alpha\rangle\ x] \\
\mathit{decodes}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle & (\mathtt{1} : x) & = [(\mathit{Inr}\ y\quad, r\ ) \mid (y\ , r\ ) \leftarrow \mathit{decodes}\ \langle\beta\rangle\ x] \\
\mathit{decodes}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle & [\,] & = [\,] \\
\mathit{decodes}\ \langle \mathrm{Prod}\ \alpha\ \beta\rangle & x & = [(y_1 \times y_2, r_2) \mid (y_1, r_1) \leftarrow \mathit{decodes}\ \langle\alpha\rangle\ x, \\
 & & \qquad\qquad\qquad\quad (y_2, r_2) \leftarrow \mathit{decodes}\ \langle\beta\rangle\ r_1]
\end{array}$$

The function is a bit more involved than *encode*, because it has to deal with incorrect input, and it has to return the unconsumed part of the input. This is solved using a standard list-of-successes parser type (Wadler 1985), where the input list is transformed into a list of pairs, containing all possible parses with the associated unconsumed parts of the input. The decoding process in not ambiguous, so only lists of zero (indicating failure) and one (indicating success) elements occur.

A value of type Unit is represented using no bits at all, therefore it can be decoded without consuming any input. The case for Sum is the only place where input is consumed (as it was the only case where output is produced in *encode*), and depending on the first bit of the input, we produce an *Inl* or an *Inr*. A third case lets the decoding process fail if we run out of input while decoding a sum. The product first decodes the left component, and then runs *decodes* for the right component on the rest of the input.

7.5 Extending the language

From a user perspective, there are almost no changes needed to allow generic functions: the syntax is unchanged, and so are the kind and type checking rules. The only thing that we have to assume is that Unit, Sum, and Prod are in the environments, together with their constructors.

What needs to be modified, though, is the translation: for datatypes, a *structural representation* is generated, replacing the toplevel structure of a datatype by a type expression constructed from the Unit, Sum, and Prod types. This is what makes it possible to reduce such a large class of types to these three datatypes.

Next to that, the translation of type-indexed functions needs to be modified. So far, we have generated components for the types in the signature of a generic function. Now, we have to generate additional components for datatypes that do not occur in the signature, but appear in the type arguments of calls to a generic function. The specialization mechanism has to be adapted to record the components that are required.

We will sketch the changes in the following, and treat them in detail in Chapters 10 and 11.

7.5.1 Structural representation of datatypes

Each datatype consists of multiple constructors, and each of the constructors has a number of fields. Constructors represent choice: a value can be constructed by means of one constructor or another, but not by both. The fields of a constructor represent pairing: the fields translate to arguments of the constructor, and if a constructor has n fields, then it could also be written as having one n-tuple as argument, or, isomorphically, a nested pair of in total n components.

The idea of the translation of datatypes is thus: replace the choice between the constructors by a nested application of Sum, and for each constructor, replace the sequence of fields by a nested application of Prod. Only if a constructor is nullary, the type Unit is used instead of the product. The fields of the constructors are *not* translated.

An example: the datatype of binary trees was defined as

$$\textbf{data}\ \text{Tree}\qquad (a :: *) = \mathit{Leaf} \mid \mathit{Node}\ (\text{Tree}\ a)\ a\ (\text{Tree}\ a)\ .$$

The translated datatype, which we will call Str(Tree), is the type synonym

$$\textbf{type}\ \mathsf{Str}(\text{Tree})\ (a :: *) = \text{Sum Unit}\ (\text{Prod}\ (\text{Tree}\ a)\ (\text{Prod}\ a\ (\text{Tree}\ a)))\ .$$

The function Str is similar to the function cp that we already know: it constructs fresh names from existing names. In this case, Str takes a type name, such as Tree, and yields a new type name, for the structural representation.

The choice between the two constructors of Tree is represented by an application of Sum in Str(Tree). The first constructor, *Leaf*, has no fields. Therefore, its representation is Unit. We apply Prod twice, nested, to translate the three fields of the second constructor *Node*. The fields itself occur again in the representation type, as does the type parameter a of Tree. In fact, if a type T is of kind κ, then $\mathsf{Str}(T)$ is of kind κ as well.

It may be surprising that the structural representation Str(Tree) still refers to the original type Tree. The structural representation is only for the toplevel of a datatype. All fields of all constructors, thus including recursive calls to the original datatype, appear in the representation type without modification. This one-layer translation turns out to be sufficient for our purposes, and it prevents problems with recursive type synonyms or infinite types.

A type and its structural representation are always *isomorphic* (if we ignore undefined values). The isomorphism is witnessed by a so-called **embedding-projection pair**, i.e., a value of the datatype

$$\textbf{data}\ \text{EP}\ (a :: *)\ (b :: *) = \mathit{EP}\ (a \to b)\ (b \to a)\ .$$

If T and $\mathsf{Str}(T)$ are indeed isomorphic, and T is of kind $\{\kappa_i \to\}^{i \in 1..n} *$, then there exists a value

$$\mathsf{ep}(T) :: \{\forall a_i :: \kappa_i.\}^{i \in 1..n}\ \text{EP}\ (T\ \{a_i\}^{i \in 1..n})\ (\mathsf{Str}(T)\ \{a_i\}^{i \in 1..n})$$

with $\mathsf{ep}(T) = \mathit{EP\ from\ to}$ such that

$$\begin{aligned} &\mathit{from} \cdot \mathit{to} &&\equiv \mathit{id} :: \{\forall a_i :: \kappa_i.\}^{i \in 1..n}\ \mathsf{Str}(T)\ \{a_i\}^{i \in 1..n} \to \mathsf{Str}(T)\ \{a_i\}^{i \in 1..n} \\ &\mathit{to} \cdot \mathit{from} &&\equiv \mathit{id} :: \{\forall a_i :: \kappa_i.\}^{i \in 1..n}\ T\ \{a_i\}^{i \in 1..n} \to T\ \{a_i\}^{i \in 1..n}\ . \end{aligned}$$

The function ep constructs again a new name, this time a function name (the name of the embedding-projection pair) from a type name.

In the case of our example, for the type Tree, the embedding-projection pair witnessing the isomorphism can be defined as follows:

$$\begin{array}{lll}
\mathsf{ep}(\mathrm{Tree}) :: \forall a :: *.\ \mathrm{EP}\ (\mathrm{Tree}\ a)\ (\mathsf{Str}(\mathrm{Tree})\ a) & & \\
\mathsf{ep}(\mathrm{Tree}) = \mathbf{let}\ \mathit{fromTree}\ \mathit{Leaf} & & = \mathit{Inl}\ \mathit{Unit} \\
\qquad\qquad\qquad\quad \mathit{fromTree}\ (\mathit{Node}\ \ell\ x\ r) & & = \mathit{Inr}\ (\ell \times (x \times r)) \\
\qquad\qquad\qquad\quad \mathit{toTree}\quad (\mathit{Inl}\ \mathit{Unit}) & & = \mathit{Leaf} \\
\qquad\qquad\qquad\quad \mathit{toTree}\quad (\mathit{Inr}\ (\ell \times (x \times r))) & & = \mathit{Node}\ \ell\ x\ r \\
\qquad\qquad\quad\ \mathbf{in}\ \ \mathit{EP}\ \mathit{fromTree}\ \mathit{toTree}\ \ . & &
\end{array}$$

In Chapter 10, we will formally define the translation of a type to its structural representation, together with the generation of a suitable embedding-projection pair. We will see that the embedding-projection pair consists of isomorphisms between the two types.

7.5.2 Specialization to datatypes not in the signature

In Chapter 6, we have established that a generic application to a type argument that is the *application* of two types is translated via

$$[\![x\ \langle t_1\ t_2\rangle]\!]^{\mathrm{tif}} \qquad \equiv [\![x\ \langle t_1\ \alpha\rangle]\!]^{\mathrm{tif}}\ \{[\![y_i\ \langle t_2\rangle]\!]^{\mathrm{tif}}\}^{i\in 1..k}\ ,$$

where $\{y_i\}^{i\in 1..k}$ are the dependencies of function x.

On the other hand, if the type argument is a named type (possibly followed by dependency variables to make it kind correct), then

$$[\![x\ \langle T\ \{\alpha_i\}^{i\in 1..n}\rangle]\!]^{\mathrm{tif}} \equiv \mathsf{cp}(x, T)\ \ .$$

For this translation to be possible, T must be in the signature of x. We will not change the translation, but lift the restriction: in the presence of structural representations of the datatypes, we can *generate* additional components, beyond the ones in the signature of a function, upon demand.

At the definition site of a generic function, we will thus statically locate all the uses of the function to determine which types occur in the type arguments of that function. If any of these types do not occur in the signature of the function, the compiler will try to generate a component. This is possible if the type is not abstract, i.e., if its definition and thus a structural representation is available. The definition of the type may contain other types, so this process may need to be recursively applied. Only if we run into an abstract type that is not in the signature, a specialization error is reported – but that should happen far less frequently than before we had generic functions.

We will now sketch how to generate a component $\mathsf{cp}(\mathit{encode}, \mathrm{Tree})$ (for function *encode* defined on page 114). Before we can define this component, we must make a detour: the structural representation of Tree t is

$$\text{Sum Unit (Prod (Tree } t\text{) (Prod } t \text{ (Tree } t\text{)))} \;,$$

and for the moment let us assume that

$$\mathit{encode}\;\Big\langle \text{Sum Unit}\;\Big(\text{Prod (Tree } \alpha\text{) }\big(\text{Prod } \alpha \text{ (Tree } \alpha\text{)}\big)\Big)\Big\rangle$$

is the call that needs to be translated. According to the rules that we know so far, and assuming that several components are in scope, the corresponding translation is

$$\begin{array}{l}
\lambda\mathsf{cp}(\mathit{encode},\alpha) \rightarrow \\
\quad \mathsf{cp}(\mathit{encode},\text{Sum}) \\
\quad\quad \mathsf{cp}(\mathit{encode},\text{Unit}) \\
\quad\quad \Big(\mathsf{cp}(\mathit{encode},\text{Prod}) \\
\quad\quad\quad \big(\mathsf{cp}(\mathit{encode},\text{Tree})\;\mathsf{cp}(\mathit{encode},\alpha)\big) \\
\quad\quad\quad \Big(\mathsf{cp}(\mathit{encode},\text{Prod})\;\mathsf{cp}(\mathit{encode},\alpha) \\
\quad\quad\quad\quad\quad\quad\quad\quad \big(\mathsf{cp}(\mathit{encode},\text{Tree})\;\mathsf{cp}(\mathit{encode},\alpha)\big)\Big)\Big) \;.
\end{array}$$

The type argument involves a dependency variable α, and *encode* depends on itself, hence the translation depends on the explicit argument that we choose to call $\mathsf{cp}(\mathit{encode},\alpha)$, accordingly. (The translation would have a different shape for a function such as *equal*, which has two dependencies.) The form of the type expression is reflected on the value level in the translation: the application of types has been replaced by application of components. We will name the above translation $\mathsf{cp}(\mathit{encode},\mathsf{Str}(\text{Tree}))$.

Note that $\mathsf{cp}(\mathit{encode},\mathsf{Str}(\text{Tree}))$ still depends on $\mathsf{cp}(\mathit{encode},\text{Tree})$. Given the declaration of $\mathsf{cp}(\mathit{encode},\mathsf{Str}(\text{Tree}))$, however, it is not difficult to come up with an implementation of $\mathsf{cp}(\mathit{encode},\text{Tree})$. After all, we have $\mathsf{ep}(\text{Tree})$ available to convert between original type and its structural representation. Let us assume that *from* and *to* are functions defined as follows:

$$\begin{array}{ll}
\mathit{from} & (\mathit{EP}\;x\;y) = x \\
\mathit{to} & (\mathit{EP}\;x\;y) = y \;.
\end{array}$$

We can then informally write

$$\mathit{encode}\;\langle\text{Tree }\alpha\rangle\;x = \mathit{encode}\;\langle\mathsf{Str}(\text{Tree})\;\alpha\rangle\;\big(\mathit{from}\;\mathsf{ep}(\text{Tree})\;x\big)$$

which, as component, gives

$$\begin{aligned}\mathsf{cp}(\mathit{encode}, \text{Tree}) = \lambda\mathsf{cp}(\mathit{encode}, \alpha) \to \lambda x \to \\ \mathsf{cp}(\mathit{encode}, \mathsf{Str}(\text{Tree}))\ \mathsf{cp}(\mathit{encode}, \alpha) \\ (\mathit{from}\ \mathsf{ep}(\text{Tree})\ x)\ .\end{aligned}$$

The two functions $\mathsf{cp}(\mathit{encode}, \text{Tree})$ and $\mathsf{cp}(\mathit{encode}, \mathsf{Str}(\text{Tree}))$ are mutually recursive, but if defined together in one recursive let, they do the job. Using partial application in a cunning way, one can see that the definition we have given for *encode* $\langle\text{Tree}\ \alpha\rangle$ is equivalent to what one would write by hand:

$$\begin{aligned}&\mathit{encode}\ \langle\text{Tree}\ \alpha\rangle\ \mathit{Leaf} = \mathtt{0} \\ &\mathit{encode}\ \langle\text{Tree}\ \alpha\rangle\ (\mathit{Node}\ \ell\ x\ r) = \\ &\quad \mathtt{1} : (\mathit{encode}\ \langle\text{Tree}\ \alpha\rangle\ \ell + \mathit{encode}\ \langle\alpha\rangle\ x + \mathit{encode}\ \langle\text{Tree}\ \alpha\rangle\ r)\ .\end{aligned}$$

The fact that we had to apply the *from* function once in the definition of the component $\mathsf{cp}(\mathit{encode}, \text{Tree})$ to convert from trees to their structural representations is connected to the base type of *encode*, which is $a \to [\text{Bit}]$. The type argument a appears once as an argument in the function. If the type would be different, the conversion would need to be different. For instance, the base type of *add* is $a \to a \to a$. Next to the two arguments, that need to be converted via *from*, also the result is of the argument type. The component for the structural representation of Tree,

$$\begin{aligned}&\mathsf{cp}(\mathit{add}, \mathsf{Str}(\text{Tree})) = \\ &\quad \lambda\mathsf{cp}(\mathit{add}, \alpha) \to \\ &\qquad \mathsf{cp}(\mathit{add}, \text{Sum}) \\ &\qquad\quad \mathsf{cp}(\mathit{add}, \text{Unit}) \\ &\qquad\quad \Big(\mathsf{cp}(\mathit{add}, \text{Prod})\ (\mathsf{cp}(\mathit{add}, \text{Tree})\ \mathsf{cp}(\mathit{add}, \alpha)) \\ &\qquad\qquad\qquad \Big(\mathsf{cp}(\mathit{add}, \text{Prod})\ \mathsf{cp}(\mathit{add}, \alpha) \\ &\qquad\qquad\qquad\qquad (\mathsf{cp}(\mathit{add}, \text{Tree})\ \mathsf{cp}(\mathit{add}, \alpha))\Big)\Big)\ ,\end{aligned}$$

has the same shape as before $\mathsf{cp}(\mathit{encode}, \mathsf{Str}(\text{Tree}))$, because *add*, as *encode*, depends on only on itself. But the type of the component is

$$\forall a :: *.\ (a \to a \to a) \to \mathsf{Str}(\text{Tree})\ a \to \mathsf{Str}(\text{Tree})\ a \to \mathsf{Str}(\text{Tree})\ a\ .$$

In consequence, while defining $\mathsf{cp}(\mathit{add}, \text{Tree})$, we not only have to convert the two incoming trees to type Str(Tree), but also the result back to the original Tree type:

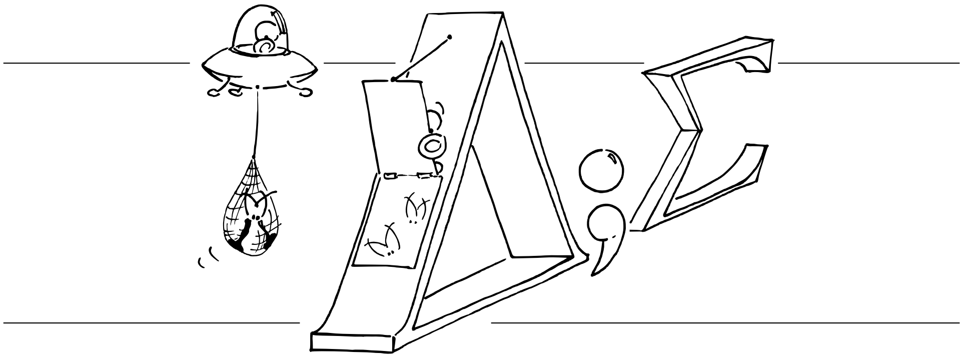
$$\begin{aligned}\mathsf{cp}(\mathit{add}, \text{Tree}) = \lambda\mathsf{cp}(\mathit{add}, \alpha) \to \lambda x \to \lambda y \to \\ \mathit{to}\ \mathsf{ep}(\text{Tree})\end{aligned}$$

$$\begin{aligned}&\Big(\mathsf{cp}(\mathit{add}, \mathsf{Str}(\mathrm{Tree}))\ \mathsf{cp}(\mathit{add}, \alpha)\\ &\quad (\mathit{from}\ \mathsf{ep}(\mathrm{Tree})\ x)\ (\mathit{from}\ \mathsf{ep}(\mathrm{Tree})\ y)\Big)\ .\end{aligned}$$

If the base type of a generic function is more complicated, the conversion required is more involved as well.

The whole process of generating components, together with the required generation of conversions between types and their structural representations, is the topic of Chapter 11.

8 Local Redefinition



In Chapter 6, we have introduced dependencies and dependency constraints. A type-indexed function can depend on other type-indexed functions. Dependencies are caused whenever one type-indexed function is called within another, with a variable type argument.

Internally, dependency constraints can be seen as hidden arguments that are made explicit by the translation. Seen in this light, dependencies play a role in two situations: during the definition of a type-indexed function, where calls to dependent functions translate into lambda abstractions that need to be supplied by the compiler whenever the component is called; and during the call of generic functions, where an application in the type argument is translated into an application of components, where the arguments required are determined by the dependencies of the function that is called.

Dependency constraints are recorded on the type level. The type checking rules of Chapter 6 make use of an environment Δ to store dependency constraints, and this environment is only modified in the arm of a generic function, where the dependencies of the generic function are added while checking the arm, and during applications of generic functions, where the entries in Δ are used to supply the arguments needed in the call.

In this chapter, we will give the programmer more control over Δ: we make it possible to locally extend or redefine the dependency environment. Dependencies are like class (Wadler and Blott 1989; Jones 1994) dictionaries: they explain how to perform a generic operation on a specific datatype. By giving the user access to the environment that stores the dictionaries, the behaviour of a generic function can be changed or extended depending on the location in the program. What this means and why it is useful is best shown by example, therefore the next sections will demonstrate different applications of the feature. Afterwards, we provide the necessary extensions to the language in Section 8.5.

8.1 Enhanced equality

For some datatypes, there are different ways to determine equality, or even different notions of equality. For lists, we can compare the lengths of the lists first and only then check the elements, and thereby save a lot of work in cases where we have to compare lists of which the prefixes are frequently identical. Other datatypes may have an amount of redundancy encoded, that is superfluous to compare or even should be ignored. For strings, which are lists of characters in Haskell, we may have applications for which it is irrelevant if something is written in upper- or lowercase letters or any mixture thereof. The string `"laMBdA"` might be considered equal to `"Lambda"` in some situations, but not in others.

The first two example cases are easy to solve with generic functions: if we decide that we want to check the length when testing lists for equality, then we must make sure that the definition of *equal* (the original definition is on page 112) has a specific case for the list type constructor [ ] that instructs the compiler to do so. The generic functionality for other datatypes is not compromised by this specific case. Similarly, if special equality algorithms are required for specific datatypes, they should also be included in the definition of the generic *equal* definition. More problematic is the latter case: we have some situations in which we want to compare case-insensitively, and others in which case-sensitivity is desired.

One solution is to define two functions, *equal* and *equalCaseInsensitive*, where the former compares strictly, and the latter ignores the case for strings. Both definitions look the same, except for the case on characters. There are a couple of disadvantages to this approach.

First, it is a lot of work. This problem can be alleviated, which is exactly what default cases are for (cf. Chapter 14).

Second, we might need a lot of different equality functions. If there are two ways to compare characters and two ways to compare floats, then there are already at least four ways to compare data structures containing floats and characters. In other words, the number of function definitions required grows exponentially with the number of different possibilities for equality (although certainly not all possibilities are used).

Third, at definition time of the generic function it may not be obvious which alternative notions of equality will be required somewhere in the program. Even a good library writer cannot possibly foresee all application areas where his library will ever be put to use, but would nevertheless like to spare the programmer the burden of defining his own generic functions.

Fourth, if there are different generic functions for the different types of equality, then there is no pragmatic way to combine them: say that only some of the strings in a data structure should be compared case-insensitively, whereas the others should be tested for standard equality. This is not possible without defining yet another generic function, thereby partially defeating the whole purpose of generic programming.

A case-insensitive equality test for characters can easily be implemented in Haskell as a monomorphic function

$$\mathit{equalCaseInsensitive}\ x\ y = \mathit{toUpper}\ x == \mathit{toUpper}\ y\ .$$

Whereas we usually would compare the strings `"laMBdA"` and `"Lambda"` using the generic call

$$\mathit{equal}\ \langle[\mathrm{Char}]\rangle\ \texttt{"laMBdA"}\ \texttt{"Lambda"}\ ,$$

which would result in *False*, we are now going to locally redefine the equality function by marking the position which is to be redefined with a dependency variable:

$$\mathit{equal}\ \langle[\alpha]\rangle\ \texttt{"laMBdA"}\ \texttt{"Lambda"}\ .$$

This call, on its own, is type incorrect, because it has unsatisfied dependencies (the dependency environment under which a complete program is checked is empty and is only filled for the right hand sides of typecase arms). It could be assigned the qualified type

$$(\mathit{enum}\ \langle[\alpha]\rangle :: [\mathrm{Char}], \mathit{equal}\ \langle[\alpha]\rangle :: \mathrm{Char} \to \mathrm{Char} \to \mathrm{Bool}) \Rightarrow \mathrm{Bool}\ ,$$

if we would allow qualified types to be assigned to simple expressions. The two dependencies to *enum* (defined on page 111) and *equal* stem from the fact that the function *equal*, as we have defined it on page 112, depends on both *enum* and *equal*.

Local redefinition allows us to locally fill in these dependencies:

$$
\begin{array}{l}
\textbf{let}\ \mathit{equal}\ \langle\alpha\rangle = \mathit{equalCaseInsensitive}\\
\quad\ \mathit{enum}\ \langle\alpha\rangle = \mathit{enum}\ \langle\mathrm{Char}\rangle\\
\textbf{in}\ \ \mathit{equal}\ \langle[\alpha]\rangle\ \texttt{"laMBdA"}\ \texttt{"Lambda"}\ .
\end{array}
$$

This expression is of type Bool, without open dependencies, and it evaluates to *True*. The function *enum* does not actually play a role, so it is safest to fall back to the default behaviour for characters.

It is important to realize that dependency variables are not instantiated by one concrete type, such as ordinary type variables usually are. Instead, we locally *name* the element type of lists with dependency variable α, and we describe how elements that occur at the position named α can be tested for equality or enumerated. But α could not simply be instantiated to Char. The call *equal* $\langle[\mathrm{Char}]\rangle$ always refers to the original definition of *equal* for lists on characters, and cannot be modified by redefinition. The expression

$$
\begin{array}{l}
\textbf{let}\ \mathit{equal}\ \langle\alpha\rangle = \mathit{equalCaseInsensitive}\\
\quad\ \mathit{enum}\ \langle\alpha\rangle = \mathit{enum}\ \langle\mathrm{Char}\rangle\\
\textbf{in}\ \ \mathit{equal}\ \langle([\alpha],[\mathrm{Char}])\rangle\ (\texttt{"laMBdA"},\texttt{"laMBdA"})\ (\texttt{"Lambda"},\texttt{"Lambda"})
\end{array}
$$

evaluates to *False*: the characters in the first components are compared using *equalCaseInsensitive* yielding *True* for this part of the equality test, but the elements in the second components are compared using the standard equality for type Char and are unaffected by the local redefinition, thus resulting in *False* for the total computation.

On the other hand, one redefinition can be used for a larger part of program text, involving multiple calls to generic functions. The next section contains an example of such a call.

Local redefinition is very similar to the binding of implicit parameters (Lewis *et al.* 2000).

8.2 Size of data structures

We will now write a truly generic *size* function, to determine the number of elements in a data structure. This definition replaces the old *size* function from page 53 that works only for a limited number of datatypes.

$$
\begin{array}{lll}
\mathit{size}\ \langle a :: *\rangle & & :: (\mathit{size}) \Rightarrow a \to \mathrm{Int}\\
\mathit{size}\ \langle\mathrm{Int}\rangle & x & = 0\\
\mathit{size}\ \langle\mathrm{Char}\rangle & x & = 0
\end{array}
$$

$$\begin{array}{lll}
size\ \langle \text{Float} \rangle & x & = 0 \\
size\ \langle \text{Unit} \rangle & Unit & = 0 \\
size\ \langle \text{Sum}\ \alpha\ \beta \rangle & (Inl\ x) & = size\ \langle \alpha \rangle\ x \\
size\ \langle \text{Sum}\ \alpha\ \beta \rangle & (Inr\ x) & = size\ \langle \beta \rangle\ x \\
size\ \langle \text{Prod}\ \alpha\ \beta \rangle & (x \times y) & = size\ \langle \alpha \rangle\ x + size\ \langle \beta \rangle\ y\ \ .
\end{array}$$

Defined like this, the *size* function seems quite a disappointment: it returns 0 on all datatypes. With local redefinition, though, we can make it useful: the code fragment

$$\begin{array}{l}
\textbf{let}\ size\ \langle \alpha \rangle = const\ 1 \\
\textbf{in}\ \ size\ \langle [\alpha] \rangle
\end{array}$$

is an expression of type $\forall a :: *.\ [a] \to \text{Int}$ and evaluates to the length of a list. We explicitly state what to count – in this case, the elements of the list. It is possible to draw a clear boundary between shape and content (Jay *et al.* 1998) by placing the α with care. The following expression demonstrates four different ways to calculate the size of a list of lists:

$$\begin{array}{lll}
\textbf{let}\ size\ \langle \alpha \rangle = const\ 1 & & \\
\textbf{in}\ \ (size\ \langle [[\text{Int}]] \rangle & [[1,2,3],[4,5]], \\
\qquad size\ \langle [[\alpha]] \rangle & [[1,2,3],[4,5]], \\
\qquad size\ \langle [\alpha] \rangle & [[1,2,3],[4,5]], \\
\qquad size\ \langle \alpha \rangle & [[1,2,3],[4,5]])\ \ .
\end{array}$$

The above expression evaluates to

$$(0,5,2,1)\ \ .$$

The first call treats the list of lists as a constant value, therefore its size is 0. The second call treats the argument as a list of lists, and thus counts all integer elements, of which there are five. The third call views the argument as a list, which has two lists as elements, therefore the result 2. Finally, the last call treats its entire argument as an element of the structure, resulting in 1 as an answer.

The example shows that it is not just a burden, but can be really useful that we are able to specify the type arguments to type-indexed and generic functions explicitly. The four calls to *size* are identical but for the type arguments involved – yet they produce four different results. This is an important observation when we discuss type inference of type arguments, in Section 13.1.

The example above demonstrates also that local redefinition via a let statement does not have to surround a call to a type-indexed function tightly, but may scope over a larger program fragment, as long as the type of the dependency provided fits everywhere where it is required.

Multiple dependency variables can be used in one type argument, and the same dependency variable may occur more than once in a type argument: the expression

$$
\begin{array}{l}
\textbf{let}\; \mathit{size}\; \langle\alpha\rangle = \mathit{const}\; 1 \\
\quad\;\; \mathit{size}\; \langle\beta\rangle = \mathit{const}\; 0 \\
\textbf{in}\;\; \mathit{size}\; \langle[\mathrm{Either}\; (\alpha,\alpha)\; \beta]\rangle \\
\qquad\quad [\mathit{Left}\; (1,2), \mathit{Right}\; (+), \mathit{Left}\; (2,4), \mathit{Right}\; (\lambda x\; y \to 0)]
\end{array}
$$

evaluates to 4, because each of the two pairs counts as 2, whereas the functions are ignored. The redefinition for β causes *size* to work on function types, which it normally does not.

If a redefinition is sufficiently polymorphic, it is possible to use the function at different types in different calls: the fragment

$$
\begin{array}{l}
\textbf{let}\; \mathit{size}\; \langle\alpha\rangle = \mathit{const}\; 1 \\
\textbf{in}\;\; (\mathit{size}\; \langle[\alpha]\rangle\; [1,2,3], \mathit{size}\; \langle[\alpha]\rangle\; [\texttt{"foo"}, \texttt{"bar"}])
\end{array}
$$

is a valid expression that evaluates to $(3,2)$.

8.3 Short notation

Very frequently, a type-indexed function depends only on one function (usually itself). In this case, the syntax for local redefinition is unnecessarily verbose. It requires to write the name of the dependency which should be redefined, but if there is only one, there is not much of a choice.

$$[\![e_{\text{FCR+tif+par}} :: t_{\text{FCR+tif+par}}]\!]^{\text{par}}_{K;\Gamma;\Delta;\Sigma} \equiv e_{\text{FCR}}$$

$$
\frac{\begin{array}{c}
K \vdash A :: \kappa_1 \to \kappa_2 \qquad \mathsf{dependencies}_\Gamma(x) \equiv y \\
\mathsf{gapp}_{K;\Gamma}(x\; \langle A\rangle) \equiv q \qquad [\![x\; \langle A\rangle]\!]^{\text{gtrans}}_{K;\Gamma;\Sigma} \equiv e \\
K;\Delta \vdash q \leqslant t
\end{array}}{[\![x\; \langle A\rangle :: t]\!]^{\text{par}}_{K;\Gamma;\Delta;\Sigma} \equiv e}\;\text{(e/tr-genapp-short)}
$$

Figure 8.1: Short notation for local redefinition

In this situation, we allow the type argument to the type-indexed function to be of higher kind. The local redefinition can then be passed as an ordinary, positional function argument.

A call such as

$$\textbf{let}\ \mathit{size}\ \langle\alpha\rangle = \mathit{const}\ 1\ \textbf{in}\ \mathit{size}\ \langle[\alpha]\rangle$$

can be abbreviated to

$$\mathit{size}\ \langle[\,]\rangle\ (\mathit{const}\ 1)\ ,$$

and

$$\begin{aligned}&\textbf{let}\ \mathit{size}\ \langle\alpha\rangle = \mathit{const}\ 1\\ &\qquad \mathit{size}\ \langle\beta\rangle = \mathit{id}\\ &\textbf{in}\ \ \mathit{size}\ \langle\mathrm{Either}\ \alpha\ \beta\rangle\end{aligned}$$

becomes alternatively

$$\mathit{size}\ \langle\mathrm{Either}\rangle\ (\mathit{const}\ 1)\ \mathit{id}\ \ .$$

Nevertheless, the explicit notation is more expressive. Local redefinitions around type-indexed function calls such as *size* $\langle\mathrm{Either}\ \alpha\ \alpha\rangle$ or *size* $\langle[[\alpha]]\rangle$ cannot easily be written using short notation.

Short notation can be defined by adding a variant of rule (e/tr-genapp) (cf. Figure 6.13) to the translation from FCR+tif+par to FCR that we have defined in Section 6.4. The original rule,

$$\frac{\begin{gathered}\mathrm{K} \vdash A :: *\\ \mathsf{gapp}_{\mathrm{K};\Gamma}(x\ \langle A\rangle) \equiv q \qquad [\![x\ \langle A\rangle]\!]^{\mathrm{gtrans}}_{\mathrm{K};\Gamma;\Sigma} \equiv e\\ \mathrm{K};\Delta \vdash q \leqslant t\end{gathered}}{[\![x\ \langle A\rangle :: t]\!]^{\mathrm{par}}_{\mathrm{K};\Gamma;\Delta;\Sigma} \equiv e}\qquad \text{(e/tr-genapp)}\ ,$$

restricts the type argument A to kind $*$. However, we have defined both gapp and $[\![\cdot]\!]^{\mathrm{gtrans}}$ in such a way that they accept type arguments of higher kind. The reason why we have disallowed such arguments in (e/tr-genapp) is that the type of such a call is a function that expects arguments for all the dependencies of the type-indexed function x. If there is more than one function, then the order in which the arguments are expected depends on the internal ordering of the dependency constraints (we suggested lexical ordering, but this is essentially irrelevant as long as there is a well-defined ordering). However, as we have noticed before, there is no possibility for confusion if x has only one dependency. Therefore, we add a different rule, (e/tr-genapp-short), in Figure 8.1. The only difference to

$$[\![e_1]\!]^{\text{short}}_{\mathrm{K};\Gamma} \equiv e_2$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash A :: \kappa_1 \to \kappa_2 \qquad \mathsf{dependencies}_\Gamma(x) \equiv y \\ \kappa_1 \equiv \{\kappa_i' \to\}^{i \in 1..n} * \\ \alpha \text{ fresh} \qquad \{\gamma_i \text{ fresh}\}^{i \in 1..n} \\ \mathsf{dependencies}_\Gamma(y) \equiv \{y\}^{k \in 1..\ell} \qquad \ell \in 0\,..\,1\end{array}}{\begin{array}{l}[\![x\ \langle A\rangle]\!]^{\text{short}}_{\mathrm{K};\Gamma} \equiv \lambda z \to \mathbf{let}\ y\ \langle \alpha\ \{(\gamma_i :: \kappa_i')\}^{i \in 1..n}\rangle = \\ \qquad\qquad z\ \{\{[\![y\ \langle \gamma_i\rangle]\!]^{\text{short}}_{\mathrm{K},\gamma_i::\kappa_i';\Gamma}\}^{k \in 1..\ell}\}^{i \in 1..n} \\ \qquad\quad \mathbf{in}\ \ [\![x\ \langle A\ \alpha\rangle]\!]^{\text{short}}_{\mathrm{K},\alpha::\kappa_1;\Gamma})\end{array}} \quad \text{(short)}$$

Figure 8.2: Alternative definition of short notation

the original rule is in the preconditions: we require the type argument to be of functional kind, and the type-indexed function that is called to have exactly one dependency. The translation is the same as before.

The correctness of the translation follows directly from Theorem 6.5, where we already have proved the more general case that the type argument may be of arbitrary kind.

Instead of defining short notation directly via extension of the translation, we could also define it as syntactic sugar for let-based local redefinition. We show this alternative definition, which makes use of a judgment of the form

$$[\![e_1]\!]^{\text{short}}_{\mathrm{K};\Gamma} \equiv e_2\ ,$$

in Figure 8.2. The judgment states that the result of translating e_1 under environments K and Γ is the expression e_2.

The only rule (short) is applicable under the condition that x has precisely one dependency, named y. The type argument A has to be of functional kind $\kappa_1 \to \kappa_2$. We introduce a fresh dependency variable α, supposed to be of kind κ_1. Depending on the arity of kind κ_1, we also introduce fresh γ_i, to serve as local arguments for α.

The translation result is a function that takes a positional argument z in place of the local redefinition. The result of the function is a let statement surrounding the recursive call $[\![x\ \langle A\ \alpha\rangle]\!]^{\text{short}}$. This call depends on $y\ \langle \alpha\ \{\gamma_i\}^{i \in 1..n}\rangle$, which is defined in the let statement to be of the form

$$z\ \{\{[\![y\ \langle \gamma_i\rangle]\!]^{\text{short}}_{\mathrm{K},\gamma_i::\kappa_i';\Gamma}\}^{k \in 1..\ell}\}^{i \in 1..n}\ .$$

This might seem more complex than expected: the function y can have no dependencies or one, but if it has one, that dependency is y again (because the dependency relation is transitive). If α is of higher kind, the function z may depend on y for all the γ_i, and is therefore supposed to be a function that takes these dependencies as explicit arguments, using short notation recursively. If α is of kind $*$, or y has no dependencies, then z takes no dependency arguments.

8.4 Local redefinition within type-indexed functions

The possibility for local redefinition is not just a useful addition to the language, but it can be absolutely necessary to have it even to define a type-indexed function.

Consider the datatype for fixpoints of functors, given by

$$\textbf{data}\ \mathrm{Fix}\ (a :: * \to *) = \mathit{In}\ (a\ (\mathrm{Fix}\ a))\ \ .$$

Using Fix, one can write recursive types in a way that makes the places of the recursion explicit. For instance, the type Fix NatF, with

$$\textbf{data}\ \mathrm{NatF}\ (a :: *) = \mathit{ZeroF} \mid \mathit{SuccF}\ a\ ,$$

is isomorphic to the type

$$\textbf{data}\ \mathrm{Nat} \qquad = \mathit{Zero} \quad \mid \mathit{Succ}\ \mathrm{Nat}$$

of natural numbers, via

$$\begin{aligned}
&\mathit{natf\_to\_nat}\ (\mathit{In}\ \mathit{ZeroF}) &&= \mathit{Zero}\\
&\mathit{natf\_to\_nat}\ (\mathit{In}\ (\mathit{SuccF}\ a)) &&= \mathit{Succ}\ (\mathit{natf\_to\_nat}\ a)\\
&\mathit{nat\_to\_natf}\ \mathit{Zero} &&= \mathit{In}\ \mathit{ZeroF}\\
&\mathit{nat\_to\_natf}\ (\mathit{Succ}\ a) &&= \mathit{In}\ (\mathit{SuccF}\ (\mathit{nat\_to\_natf}\ a))\ \ .
\end{aligned}$$

The fixpoint view of a datatype has the advantage that operations that work with the recursive structure, such as cata- and anamorphisms, can be defined easily. Fixpoints will therefore be our topic again later, in Section 12.3 and Section 17.2.

Fortunately, in most cases, we do not have to care how a generic function can be defined for datatypes such as Fix. The function is generic, and the compiler derives the definition of a call such as *equal* ⟨Fix NatF⟩ automatically. We could also define it ourselves:

$$\mathit{equal}\ \langle \mathrm{Fix}\ \alpha\rangle\ (\mathit{In}\ x)\ (\mathit{In}\ y) = \mathit{equal}\ \langle \alpha\ (\mathrm{Fix}\ \alpha)\rangle\ x\ y\ \ .$$

But what if we want to change this definition, and do something out of the ordinary? We might only be interested in equality at the top level of a datatype, and want to stop comparing once we hit a point of recursion. We can only do this using local redefinition:

$$\begin{aligned} \mathit{equal}\ \langle \mathrm{Fix}\ \alpha\rangle\ (\mathit{In}\ x)\ (\mathit{In}\ y) = \mathbf{let}\ & \mathit{equal}\ \langle\beta\rangle\ x\ y = \mathit{True} \\ & \mathit{enum}\ \langle\beta\rangle \quad\ \ = \mathit{enum}\ \langle \mathrm{Fix}\ \alpha\rangle \\ \mathbf{in}\ \ & \mathit{equal}\ \langle\alpha\ \beta\rangle\ . \end{aligned}$$

A similar situation arises whenever we want to define a type-indexed function explicitly for a type constructor which takes arguments of higher kind.

8.5 Core language with local redefinition FCR+tif+par+lr

The language extension that is needed to support local redefinition is relatively simple. We extend the syntax for value declarations with one additional case, as shown in Figure 8.3.

Value declarations

$$\begin{aligned} d \ ::=\ & \ldots && \text{everything from Figure 4.1} \\ \mid\ & x\ \langle\alpha\ \{(\gamma_i :: \kappa_i)\}^{i\in 1..n}\rangle = e && \\ & && \text{local redefinition} \end{aligned}$$

Figure 8.3: Core language with local redefinition FCR+tif+par+lr, extends language FCR+tif+par in Figures 6.1, 4.1, and 3.1

We then need a new declaration translation rule, along the lines of the rules of Figure 6.15. Since local redefinition affects the dependency and kind environments, we have to extend the form of the judgment to be able to redefine these two environments:

$$[\![d_{\text{FCR+tif+par+lr}} \rightsquigarrow K_2; \Gamma_2; \Delta_2; \Sigma_2]\!]^{\text{par}}_{K_1;\Gamma_1;\Delta_1;\Sigma_1} \equiv \{d_{\text{FCR}}\}_{;}^{i\in 1..n}\ .$$

The environments K_2 and Δ_2 have been added. The rule for recursive let, (e/tr-let), is adapted to tie the knot for these environments as it already did for Γ and Σ. The updated rule is shown in Figure 8.4.

The translation of a local redefinition declaration is described in Figure 8.5. We only allow local redefinition for a type-indexed function x that is in scope. Based on the kinds of the local arguments γ_i, we determine the kind κ of α.

$$[\![e_{\text{FCR+tif+par+lr}} :: t_{\text{FCR+tif+par+lr}}]\!]^{\text{par}}_{\mathrm{K};\Gamma;\Delta;\Sigma} \equiv e_{\text{FCR}}$$

$$\frac{\begin{array}{c} \mathrm{K}' \equiv \mathrm{K}\ \{,\mathrm{K}_i\}^{i\in 1..n} \qquad \Gamma' \equiv \Gamma\ \{,\Gamma_i\}^{i\in 1..n} \\ \Delta' \equiv \Delta\ \{,\Delta_i\}^{i\in 1..n} \qquad \Sigma' \equiv \Sigma\ \{,\Sigma_i\}^{i\in 1..n} \\ \left\{ [\![d_i \rightsquigarrow \mathrm{K}_i;\Gamma_i;\Delta_i;\Sigma_i]\!]^{\text{par}}_{\mathrm{K}';\Gamma';\Delta';\Sigma'} \equiv \{d_{i,j}\}^{j\in 1..m_i} \right\}^{i\in 1..n} \\ [\![e :: t]\!]^{\text{par}}_{\mathrm{K}';\Gamma';\Delta';\Sigma'} \equiv e' \end{array}}{[\![\mathbf{let}\ \{d_i\}^{i\in 1..n}_{;}\ \mathbf{in}\ e :: t]\!]^{\text{par}}_{\mathrm{K};\Gamma;\Delta;\Sigma} \equiv \mathbf{let}\ \left\{ \{d_{i,j}\}^{j\in 1..m_i}_{;} \right\}^{i\in 1..n}_{;}\ \mathbf{in}\ e'} \quad \text{(e/tr-let-lr)}$$

Figure 8.4: New rule for checking and translating recursive let in fcr+tif+par+lr, replaces rule (e/tr-let) in Figure 6.13

$$[\![d_{\text{FCR+tif+par+lr}} \rightsquigarrow \mathrm{K}_2;\Gamma_2;\Delta_2;\Sigma_2]\!]^{\text{par}}_{\mathrm{K}_1;\Gamma_1;\Delta_1;\Sigma_1} \equiv \{d_{\text{FCR}}\}^{i\in 1..n}_{;}$$

$$\frac{\begin{array}{c} x\ \langle a :: * \rangle :: \sigma \in \Gamma \\ \kappa \equiv \{\kappa_i \rightarrow\}^{i\in 1..n} * \\ \mathrm{K}' \equiv \mathrm{K}\ \{,\gamma_i :: \kappa_i\}^{i\in 1..n} \\ [\![e :: q]\!]^{\text{par}}_{\mathrm{K}';\Gamma;\Delta;\Sigma} \equiv e' \\ \Delta' \equiv x\ \langle \alpha\ \{(\gamma_i :: \kappa_i)\}^{i\in 1..n} \rangle :: q \end{array}}{[\![x\ \langle \alpha\ \{(\gamma_i :: \kappa_i)\}^{i\in 1..n} \rangle = e \rightsquigarrow \alpha :: \kappa; \varepsilon; \Delta'; \varepsilon]\!]^{\text{par}}_{\mathrm{K};\Gamma;\Delta;\Sigma} \equiv \mathsf{cp}(x,\alpha) = e'} \quad \text{(d/tr-lr)}$$

Figure 8.5: Translation of fcr+tif+par+lr declarations to fcr, extends Figure 6.15

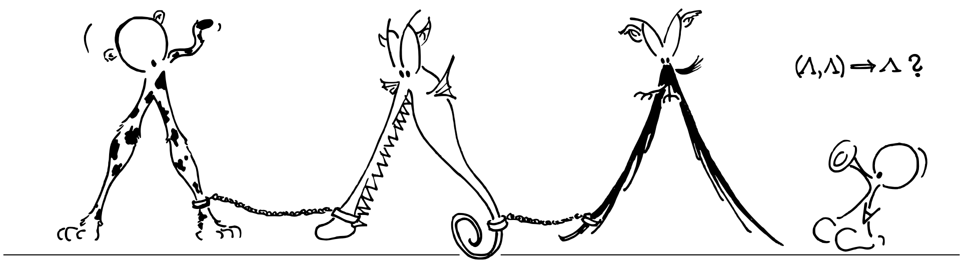
The dependency variable α is visible in the scope of the let that contains the local redefinition, therefore the new kind binding $\alpha :: \kappa$ is also exported. The right hand side expression e is checked to be of a qualified type q, and this type is added as type of the dependency constraint for $x\ \langle\alpha\ \{\gamma_i :: \kappa_i\}^{i \in 1..n}\rangle$. The translation of the declaration is consequently a declaration for $\mathtt{cp}(x, \alpha)$.

It is noteworthy that we do not make use of the type of x while checking the declaration. Restrictions on the type are only imposed by the usages of x in the scope of the let that contains the local redefinition. In other words, if the body of that let statements does not depend on $x\ \langle\alpha\ \{\gamma_i :: \kappa_i\}^{i \in 1..n}\rangle$, there are no restrictions on the type of the right hand side. For instance, the expression

```
let size ⟨α⟩ = "foo"
in  1
```

is legal and evaluates to 1.

9 Types of Type-indexed Functions



We have imposed relatively strict conditions on the types of type-indexed functions. While a large class of type-indexed and generic functions can be written within these limits, some other – very useful – functions cannot.

Luckily for the programmer, it is not a big step to surpass these limits and write functions such as *map*, *zipWith*, and *concat*, all of which will be introduced throughout the next few sections.

After the examples, we will have a look at the implications for the theory of dependencies and qualified types, which becomes considerably more complex due to the additional generality.

9.1 Identity and mapping

Let us look at the generic identity function *gid*, which can be defined as follows:

$$\begin{array}{lll} gid\ \langle a :: *\rangle & & :: (gid) \Rightarrow a \rightarrow a \\ gid\ \langle \mathrm{Int}\rangle & x & = x \end{array}$$

$$\begin{array}{lll}
\mathit{gid}\ \langle \mathrm{Char} \rangle & x & = x \\
\mathit{gid}\ \langle \mathrm{Float} \rangle & x & = x \\
\mathit{gid}\ \langle \mathrm{Unit} \rangle & x & = x \\
\mathit{gid}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inl}\ x) & = \mathit{Inl}\ (\mathit{gid}\ \langle \alpha \rangle\ x) \\
\mathit{gid}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inr}\ x) & = \mathit{Inr}\ (\mathit{gid}\ \langle \beta \rangle\ x) \\
\mathit{gid}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle & (x \times y) & = \mathit{gid}\ \langle \alpha \rangle\ x \times \mathit{gid}\ \langle \beta \rangle\ y\ .
\end{array}$$

Note that we have made a choice in the above definition: the definition is written recursively, applying the generic identity deeply to all parts of a value. We could have written the following instead of the last three lines, removing the dependency of *gid* on itself:

$$\begin{array}{l}
\mathit{gid}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle\ x = x \\
\mathit{gid}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle\ x = x\ .
\end{array}$$

But retaining the dependency and applying the function recursively has an advantage: local redefinition (cf. Chapter 8) can be used to change the behaviour of the function!

As an example, we could increase all elements of a list by one, using the function

$$\mathit{incBy1}\ x = \mathbf{let}\ \mathit{gid}\ \langle \alpha \rangle = (+1)\ \mathbf{in}\ \mathit{gid}\ \langle [\alpha] \rangle\ x\ .$$

Note that this is something that would normally be written as an application of *map*:

$$\mathit{incBy1}\ x = \mathit{map}\ (+1)\ x\ .$$

If we compare *map* with the locally redefined version of *gid*, then two differences spring to mind. First, the function *map* can only be used on lists, whereas *gid* can be used on other data structures as well. Second, *map* has a more liberal type. If we define

$$\mathit{map}'\ f = \mathbf{let}\ \mathit{gid}\ \langle \alpha \rangle = f\ \mathbf{in}\ \mathit{gid}\ \langle [\alpha] \rangle\ ,$$

then we can observe that *map′*, compared to *map* has a restricted type:

$$\begin{array}{lll}
\mathit{map}' & :: \forall a :: *. & (a \to a) \to [a] \to [a] \\
\mathit{map} & :: \forall (a :: *)\ (b :: *). & (a \to b) \to [a] \to [b]\ .
\end{array}$$

The function passed to *map* may change the type of its argument; the function passed to *map′* must preserve the argument type.

Inspired by this deficiency, we can ask ourselves if it would not be possible to also pass a function of type $a \to b$ while locally redefining *gid*. The function $\mathit{gid}\ \langle [\alpha] \rangle$ has the qualified type

$$gid\ \langle[\alpha]\rangle\ ::\ \forall a :: *.\qquad (gid\ \ \langle\alpha\rangle :: a \to a) \Rightarrow [a] \to [a]\ ,$$

but we are now going to generalize this type to

$$map\ \langle[\alpha]\rangle :: \forall (a :: *)\ (b :: *).\ (map\ \langle\alpha\rangle :: a \to b) \Rightarrow [a] \to [b]\ ,$$

thereby renaming function *gid* to function *map* (but using exactly the same definition). For this to work, *map* needs a different type signature:

$$map\ \langle a :: *, b :: *\rangle :: (map\ \langle a, b\rangle) \Rightarrow a \to b\ \ .$$

The function is now parametrized over *two* type variables, and so is the dependency. (We will discuss further extensions to the syntax of type signatures and their exact meaning later in this chapter.) When used at a constant type, both variables a and b are instantiated to the same type – only when locally redefining the function for a dependency variable, the additional flexibility is available. We will adapt the algorithm gapp from Figure 6.8 to handle type signatures such as the one of *map*, and can then use this adapted algorithm to derive types for applications of *map* to specific type arguments. Some examples of such types are shown in Figure 9.1. In all of the examples, the type argument A is assumed not to contain any free dependency variables.

$$\begin{array}{lll}
map\ \langle A :: *\rangle & :: & A \to A \\
map\ \langle A\ (\alpha :: *) :: *\rangle & :: & \forall (a_1 :: *)\ (a_2 :: *). \\
 & & \quad (map\ \langle\alpha\rangle :: a_1 \to a_2) \\
 & & \qquad \Rightarrow A\ a_1 \to A\ a_2\ , \\
map\ \langle A\ (\alpha :: *)\ (\beta :: *) :: *\rangle & :: & \forall (a_1 :: *)\ (a_2 :: *)\ (b_1 :: *)\ (b_2 :: *). \\
 & & \quad (map\ \langle\alpha\rangle :: a_1 \to a_2, \\
 & & \quad\ map\ \langle\beta\rangle :: b_1 \to b_2) \\
 & & \qquad \Rightarrow A\ a_1\ a_2 \to A\ b_1\ b_2 \\
map\ \langle A\ (\alpha :: * \to *) :: *\rangle & :: & \forall (a_1 :: * \to *)\ (a_2 :: * \to *). \\
 & & \quad (map\ \langle\alpha\ \gamma\rangle :: \forall (c_1 :: *)\ (c_2 :: *). \\
 & & \qquad\qquad (map\ \langle\gamma\rangle :: c_1 \to c_2) \\
 & & \qquad\qquad\quad \Rightarrow a_1\ c_1 \to a_2\ c_2) \\
 & & \qquad \Rightarrow A\ a_1 \to A\ a_2\ \ .
\end{array}$$

Figure 9.1: Example types for generic applications of *map* to type arguments of different form

Using the short notation of Section 8.3 can be a big help with *map*, as it allows us to use *map* almost as the standard list map, only on far more datatypes. The expressions

$$\begin{array}{llll} \mathit{map}\ \langle[\,]\rangle & (+1) & & [1,2,3,4,5] \\ \mathit{map}\ \langle(,)\rangle & (*2) & (\texttt{"y"}+\!\!+) & (21,\texttt{"es"}) \\ \mathit{map}\ \langle\text{Either}\rangle & \mathit{not} & \mathit{id} & (\mathit{Left\ True}) \end{array}$$

evaluate to $[2,3,4,5,6]$, $(42,\texttt{"yes"})$, and *Left False*, respectively.

We will consider some more examples of generalized type signatures in the next sections, before we discuss the (unfortunately) considerable theoretical implications of this generalization in Section 9.6.

9.2 Zipping data structures

The Haskell standard prelude contains a list function *zipWith* of type

$$\forall (a :: *)\ (b :: *)\ (c :: *).\ (a \to b \to c) \to [a] \to [b] \to [c]\ \ .$$

It takes a function to combine a value of type a and one of type b into a value of type c, and uses this function to combine two lists of same length pointwise. (In fact, it discards the extraneous elements of the longer list, but we will assume that it should only work on lists of the same length.) The probably better known function *zip* is an instance of *zipWith*:

$$\begin{array}{l} \mathit{zip} :: \forall (a :: *)\ (b :: *).\ [a] \to [b] \to [(a,b)] \\ \mathit{zip} = \mathit{zipWith}\ (,)\ \ . \end{array}$$

We are going to generalize the function *zipWith* to a generic function in a very similar way as we just have generalized the list function *map* to a generic function.

The trick is once again to allow more than one type variable parameter in the type signature: this time, we need even three type variables!

$$\begin{array}{lllll} \mathit{zipWith}\ \langle a :: *, b :: *, c :: * \rangle & & & & :: (\mathit{zipWith}\ \langle a,b,c\rangle) \Rightarrow a \to b \to c \\ \mathit{zipWith}\ \langle \text{Int} \rangle & x & y & & \\ \quad \mid x \equiv y & & & & = x \\ \quad \mid \mathit{otherwise} & & & & = \\ & & & \multicolumn{2}{l}{\mathit{error}\ \texttt{"args must have same shape!"}} \\ \mathit{zipWith}\ \langle \text{Unit} \rangle & \mathit{Unit} & \mathit{Unit} & & = \mathit{Unit} \\ \mathit{zipWith}\ \langle \text{Sum}\ \alpha\ \beta \rangle & (\mathit{Inl}\ x) & (\mathit{Inl}\ y) & & = \mathit{zipWith}\ \langle\alpha\rangle\ x\ y \\ \mathit{zipWith}\ \langle \text{Sum}\ \alpha\ \beta \rangle & (\mathit{Inr}\ x) & (\mathit{Inr}\ y) & & = \mathit{zipWith}\ \langle\beta\rangle\ x\ y \\ \mathit{zipWith}\ \langle \text{Sum}\ \alpha\ \beta \rangle & \_ & \_ & & = \\ & & & \multicolumn{2}{l}{\mathit{error}\ \texttt{"args must have same shape!"}} \\ \mathit{zipWith}\ \langle \text{Prod}\ \alpha\ \beta \rangle & (x_1 \times x_2) & (y_1 \times y_2) & & = \\ & & & \multicolumn{2}{l}{\mathit{zipWith}\ \langle\alpha\rangle\ x_1\ y_1 \times \mathit{zipWith}\ \langle\beta\rangle\ x_2\ y_2} \end{array}$$

Applied to type arguments without dependency variables, the function is, much as *map*, almost useless: it tests whether two values are of the same shape. If they are not, a runtime error is thrown.

But using local redefinition, the functionality of the list function *zipWith* can be recovered, but of course, we can *zip* not only lists now: the expression

$$\begin{array}{ll}\textbf{let}\ \mathit{zipWith}\ \langle\alpha\rangle = (+) & \\ \textbf{in}\ \ \mathit{zipWith}\ \langle\mathrm{Tree}\ \alpha\rangle & (\mathit{Node}\ (\mathit{Node}\ \mathit{Leaf}\ 1\ \mathit{Leaf})\ 2\ \mathit{Leaf}) \\ & (\mathit{Node}\ (\mathit{Node}\ \mathit{Leaf}\ 3\ \mathit{Leaf})\ 4\ \mathit{Leaf})\end{array}$$

evaluates to

$$\mathit{Node}\ (\mathit{Node}\ \mathit{Leaf}\ 4\ \mathit{Leaf})\ 6\ \mathit{Leaf}\ \ .$$

Figure 9.2 contains examples of types for a few specific applications of *zipWith* to type arguments of different shape, corresponding to the types for *map* given in Figure 9.1.

$$\begin{array}{ll}
\mathit{zipWith}\ \langle A :: *\rangle & ::\qquad A \to A \to A \\
\mathit{zipWith}\ \langle A\ (\alpha :: *) :: *\rangle & :: \forall (a_1 :: *)\ (a_2 :: *)\ (a_3 :: *). \\
& \quad (\mathit{zipWith}\ \langle\alpha\rangle :: a_1 \to a_2 \to a_3) \\
& \qquad \Rightarrow A\ a_1 \to A\ a_2 \to A\ a_3 \\
\mathit{zipWith}\ \langle A\ (\alpha :: *)\ (\beta :: *) :: *\rangle & :: \forall (a_1 :: *)\ (a_2 :: *)\ (a_3 :: *) \\
& \quad (b_1 :: *)\ (b_2 :: *)\ (b_3 :: *). \\
& \quad (\mathit{zipWith}\ \langle\alpha\rangle :: a_1 \to a_2 \to a_3, \\
& \quad\ \mathit{zipWith}\ \langle\beta\rangle :: b_1 \to b_2 \to b_3) \\
& \qquad \Rightarrow A\ a_1\ b_1 \to A\ a_2\ b_2 \to A\ a_3\ b_3 \\
\mathit{zipWith}\ \langle A\ (\alpha :: * \to *) :: *\rangle & :: \forall (a_1 :: * \to *)\ (a_2 :: * \to *)\ (a_3 :: * \to *). \\
& \quad (\mathit{zipWith}\ \langle\alpha\ \gamma\rangle :: \\
& \qquad \forall (c_1 :: *)\ (c_2 :: *)\ (c_3 :: *). \\
& \qquad\quad (\mathit{zipWith}\ \langle\gamma\rangle :: c_1 \to c_2 \to c_3) \\
& \qquad\qquad \Rightarrow a_1\ c_1 \to a_2\ c_2 \to a_3\ c_3) \\
& \qquad \Rightarrow A\ a_1 \to A\ a_2 \to A\ a_3\ \ .
\end{array}$$

Figure 9.2: Example types for generic applications of *zipWith* to type arguments of different form

9.3 Generically collecting values

Let us look at another function that is only useful in the context of local redefinition, but also requires that we again introduce some new syntax for the type signatures of type-indexed functions. The generic function *collect* is supposed to collect values from a data structure:

$$\begin{array}{lll}
\mathit{collect}\ \langle a :: * \mid c :: * \rangle & & :: (\mathit{collect}\ \langle a \mid c \rangle) \Rightarrow a \to [c] \\
\mathit{collect}\ \langle \mathrm{Int} \rangle & x & = [\,] \\
\mathit{collect}\ \langle \mathrm{Unit} \rangle & \mathit{Unit} & = [\,] \\
\mathit{collect}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inl}\ x) & = \mathit{collect}\ \langle \alpha \rangle\ x \\
\mathit{collect}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inr}\ x) & = \mathit{collect}\ \langle \beta \rangle\ x \\
\mathit{collect}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle & (x_1 \times x_2) & = \mathit{collect}\ \langle \alpha \rangle\ x_1 +\!\!+ \mathit{collect}\ \langle \beta \rangle\ x_2
\end{array}$$

We will discuss the type signature below, and first discuss the implementation: the function is defined in a way that always returns the empty list on constant type arguments. For kind $*$ types such as Int or Unit, the empty list is returned directly. In a sum, we descend to the branch that is selected by the argument. For a product, we concatenate the resulting lists of the two components. The only way to add content to the list is by local redefinition.

The type signature of *collect* is interesting. Once more, the type is parametrized over more than one type variable. However, the c is different from the type parameters we have seen so far as it appears to the right of a vertical bar. The c is called a *non-generic* type variable. Such non-generic variables appear in type-indexed functions that are *parametrically polymorphic* with respect to some type variables. The *collect* function is, as defined, parametrically polymorphic in the element type of its list result. It returns always the empty list, so there is no need, but also no desire, to fix the type of the list elements.

We can further specify this type as soon as we use *collect* in a local redefinition. Let us look at specific example cases for the type of *collect* again, presented in Figure 9.3, to get an intuition about how the non-generic quantified variable influences the type. As can easily be seen, the variable c is always quantified just once, at the outside, and is used in all the dependency constraints (also nested ones) without modification.

As mentioned before, we can make use of *collect* through local redefinition. For example, the list function *concat* is

$$\mathbf{let}\ \mathit{collect}\ \langle \alpha \rangle\ x = x\ \mathbf{in}\ \mathit{collect}\ \langle [\alpha] \rangle\ ,$$

whereas

$$\mathbf{let}\ \mathit{collect}\ \langle \alpha \rangle\ x = [x]\ \mathbf{in}\ \mathit{collect}\ \langle \mathrm{Tree}\ \alpha \rangle$$

computes the inorder traversal of a tree. But also other applications of *collect* are possible: all negative list elements can be collected by

$$
\begin{array}{ll}
\mathit{collect}\ \langle A :: *\rangle :: \forall c :: *. & A \to [c] \\
\mathit{collect}\ \langle A\ \langle \alpha :: *\rangle :: *\rangle & \\
\qquad :: \forall (a :: *)\ (c :: *). & (\mathit{collect}\ \langle \alpha\rangle :: a \to [c]) \\
& \quad \Rightarrow A\ a \to [c] \\
\mathit{collect}\ \langle A\ \langle \alpha :: *\rangle\ (\beta :: *) :: *\rangle & \\
\qquad :: \forall (a :: *)\ (b :: *)\ (c :: *). & (\mathit{collect}\ \langle \alpha\rangle :: a \to [c], \\
& \ \mathit{collect}\ \langle \beta\rangle :: b \to [c]) \\
& \quad \Rightarrow A\ a\ b \to [c] \\
\mathit{collect}\ \langle A\ \langle \alpha :: * \to *\rangle :: *\rangle & \\
\qquad :: \forall (a :: * \to *)\ (c :: *). & (\mathit{collect}\ \langle \alpha\ \gamma\rangle :: \\
& \qquad \forall b :: *. (\mathit{collect}\ \langle \gamma\rangle :: b \to [c]) \\
& \qquad\qquad \Rightarrow a\ b \to [c]) \\
& \quad \Rightarrow A\ a \to [c]\ .
\end{array}
$$

Figure 9.3: Example types for generic applications of *collect* to type arguments of different form

$$
\begin{array}{l}
\mathbf{let}\ \mathit{collect}\ \langle \alpha\rangle\ x \mid x < 0 \qquad = [x] \\
\qquad\qquad\qquad\quad \mid \mathit{otherwise} = [\,] \\
\mathbf{in}\ \ \mathit{collect}\ \langle [\alpha]\rangle\ .
\end{array}
$$

At a first glance, it might be unclear why we need to add the notion of *non-generic type parameters* to our language. After all, the definition of *collect* as given above would also type check with the following type signature

$$\mathit{collect}\ \langle a :: *\rangle :: (\mathit{collect}\ \langle a\rangle) \Rightarrow \forall c :: *.\ a \to [c]\ .$$

Here, there is only one type parameter, and the c is quantified on the right hand side. But although the *definition* is still correct with respect to this type signature, the resulting function cannot be used as before in the context of local redefinition. If we expand this internally quantified type for *collect* $\langle [\alpha]\rangle$, we get

$$\forall a :: *.\ (\mathit{collect}\ \langle \alpha\rangle :: \forall c :: *.\ a \to [c]) \Rightarrow \forall c :: *.\ [a] \to [c]\ .$$

If the dependency constraint itself is universally quantified, then we can only redefine the dependency to a sufficiently polymorphic function. The *concat* definition,

$$\mathbf{let}\ \mathit{collect}\ \langle \alpha\rangle\ x = x\ \mathbf{in}\ \mathit{collect}\ \langle [\alpha]\rangle\ ,$$

would fail because $\lambda x \to x$ is of type $\forall c :: *.\ c \to c$, which does not match $\forall c :: *.\ a \to [c]$. We are forced to redefine *collect* in a way that it still returns a polymorphic list, i.e., it must still return the empty list …

With the original and correct type for *collect*, we have

$$\forall(a :: *)\ (c :: *).\ (collect\ \langle\alpha\rangle :: a \to [c]) \Rightarrow [a] \to [c]$$

as type for *collect* $\langle[\alpha]\rangle$, and here c can be instantiated to be the same as a.

9.4 More choice

The generalizations of the type signatures of type-indexed functions that we have introduced in this chapter so far lead to additional complexity in the theory. One of the reasons is that functions can depend on each other in more than just one way now. Therefore, we add variables also to the list of dependencies, such as in the type signature for *map*,

$$map\ \langle a :: *, b :: *\rangle :: (map\ \langle a, b\rangle) \Rightarrow a \to b\ ,$$

where we now use *map* $\langle a, b\rangle$ to specify the dependency, instead of just *map*. We will look at a few examples that demonstrate why this is necessary.

Assume we have a list-based implementation for sets: defined as follows:

$$\mathbf{data}\ \mathrm{Set}\ (a :: *) = Set\ [a]\ \ .$$

This datatype is supposed to fulfill the invariant that no duplicates occur in the list. The generic function *map* does not respect the invariant, so it is advisable to add a specific arm to the *map* function for sets:

$$map\ \langle \mathrm{Set}\ \alpha\rangle\ (Set\ xs) = Set\ (nubBy\ (equal\ \langle\alpha\rangle)\ (map\ \langle[\alpha]\rangle\ xs))\ \ .$$

The function *nubBy* is defined in the Haskell standard library and has the type signature

$$nubBy :: \forall a :: *.\ (a \to a \to \mathrm{Bool}) \Rightarrow [a] \to [a]\ \ .$$

It removes duplicates from a list, given an equality function for the list element type. In above definition of *map* for datatype Set, we first apply *map* to the underlying list implementation, then use *nubBy* with argument *equal* $\langle\alpha\rangle$ to remove duplicates, and finally apply the *Set* constructor again to create the resulting set.

The implementation is fine, but it obviously introduces a dependency of function *map* on function *equal*. Furthermore, because *equal* depends on *enum*, the function *map* must depend on *enum* as well. The correct type signature for *map* with the additional arm for Set is thus

$$map\ \langle a :: *, b :: *\rangle :: (map\ \langle a, b\rangle, equal\ \langle b\rangle, enum\ \langle b\rangle) \Rightarrow a \to b\ \ .$$

We are in a situation where *map*, which is parametrized over two type variables, depends on functions which are parametrized over only one type variable. Therefore, there is choice. In this case, we need the equality and thus enumeration on the result type of the mapping, because we apply $\mathit{nubBy}\ (\mathit{equal}\ \langle\alpha\rangle)$ after mapping over the list.

Another (contrived) example to demonstrate the improved expressivity is the following arm of a type-indexed function that performs some sort of consistency check on a value. We just present one arm here, for a datatype of pairs, defined as

$$\mathbf{data}\ \mathrm{Pair}\ (a :: *) = \mathit{Pair}\ a\ a\ \ .$$

The type signature and the arm itself are as follows:

$$\begin{aligned}
&\mathit{check}\ \langle a :: *\rangle && :: (\mathit{map}\ \langle a, a\rangle, \mathit{collect}\ \langle a \mid \mathrm{Int}\rangle) \Rightarrow a \to \mathrm{Bool}\\
&\mathit{check}\ \langle \mathrm{Pair}\ \alpha\rangle\ (\mathit{Pair}\ x_1\ x_2) && = \mathit{sum}\ \big(\mathit{collect}\ \langle \mathrm{Pair}\ \alpha\rangle\\
& && \qquad\qquad (\mathit{Pair}\ (\mathit{map}\ \langle\alpha\rangle\ x_1)\ x_2)\big) \equiv 0\ \ .
\end{aligned}$$

The function *check* is parametrized over only one type parameter a, but it depends on both *map* and *collect*. The mapping is requested in a version that does not modify the type, and the result type of *collect* is already fixed to Int by the example, because of the usage of function $\mathit{sum} :: [\mathrm{Int}] \to \mathrm{Int}$ on the result of a call to *collect*.

In general, if a function with n type parameters depends on a function with m type parameters, there are n^m possibilities to choose the type of the dependency. In addition, if there are non-generic type parameters such as with *collect*, those can be instantiated to any type of matching kind.

To put everything in a common setting, we will now also write type signatures of one-parameter type-indexed functions using variables on the list of dependencies, for example:

$$\begin{aligned}
&\mathit{equal}\ \ \langle a :: *\rangle :: (\mathit{enum}\ \langle a\rangle, \mathit{equal}\ \langle a\rangle) \Rightarrow a \to a \to \mathrm{Bool}\\
&\mathit{empty}\ \ \langle a :: *\rangle :: (\mathit{empty}\ \langle a\rangle) \Rightarrow a\\
&\mathit{encode}\ \langle a :: *\rangle :: (\mathit{encode}\ \langle a\rangle) \Rightarrow a \to [\mathrm{Bit}]\ \ .
\end{aligned}$$

The following section will start the formal discussion of generalized type signatures.

9.5 Type tuples

As we have seen in the examples, we parametrize type signatures (and consequently, the types) of type-indexed functions no longer over just one argument,

Type tuple patterns
$\pi \quad ::= \{a_i :: *\}_{,}^{i\in 1..r} \mid \{b_j :: \kappa_j\}_{,}^{j\in 1..s}$
type tuple pattern

Type tuples
$\vartheta \quad ::= \{t_i\}_{,}^{i\in 1..r} \mid \{t'_j\}_{,}^{j\in 1..s}$ type tuple

Type argument tuples
$\Theta \quad ::= \{A_i\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s}$ type argument tuple

Figure 9.4: Syntax of type tuples and their kinds

but over a a tuple made up from two different sorts of variables: a **type tuple pattern** consisting of r **generic** and s **non-generic** variables has the form

$$\{a_i :: *\}_{,}^{i\in 1..r} \mid \{t_j :: \kappa_j\}_{,}^{j\in 1..s} \ .$$

All variables are supposed to be different. Each type variable is associated with a kind, but generic type variables are always of kind $*$. We also say that the tuple is $\langle r \mid s\rangle$**-ary**. We omit the vertical bar if $s \equiv 0$.

A type with multiple arguments must also be instantiated to multiple types, and those types must match in number and kind. A **type tuple** is of the form

$$\{t_i\}_{,}^{i\in 1..r} \mid \{t'_j\}_{,}^{j\in 1..s} \ .$$

It has the structure of a type tuple pattern, but consists of r types in the **generic slots**, and s types in the **non-generic slots**.

A **type argument tuple** is like a type tuple, but the generic slots contain type arguments instead of types:

$$\{A_i\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s} \ .$$

Figure 9.4 shows the new syntactic categories. We use π to abbreviate type tuple patterns, ϑ to denote type tuples, and Θ for type argument tuples.

We can use the kind information that is contained in a type tuple pattern to kind check a type tuple (or a type argument tuple) under a kind environment K, using judgments of the forms

$$\mathsf{K} \vdash^{pat} \vartheta :: \pi \rightsquigarrow \varphi$$
$$\mathsf{K} \vdash^{pat} \Theta :: \pi \rightsquigarrow \varphi \ ,$$

where φ is a substitution mapping the variables that occur in pattern π to the respective types or type arguments in ϑ or Θ. The rules are shown in Figure 9.5.

$$\mathrm{K} \vdash^{pat} \vartheta :: \pi \rightsquigarrow \varphi$$

$$\frac{\begin{array}{c}\{\mathrm{K} \vdash t_i :: \kappa\}^{i \in 1..r} \qquad \{\mathrm{K} \vdash t'_j :: \kappa_j\}^{j \in 1..s} \\ \varphi \equiv \{(a_i \mapsto t_i)\}^{i \in 1..r}_{\cdot} \; \{\cdot (b_j \mapsto t'_j)\}^{j \in 1..s}\end{array}}{\mathrm{K} \vdash^{pat} \left(\{t_i\}^{i \in 1..r}_{,} \mid \{t'_j\}^{j \in 1..s}_{,}\right) :: \left(\{a_i :: \kappa\}^{i \in 1..r}_{,} \mid \{b_j :: \kappa_j\}^{j \in 1..s}_{,}\right) \rightsquigarrow \varphi} \quad \text{(tt)}$$

$$\mathrm{K} \vdash^{pat} \Theta :: \pi \rightsquigarrow \varphi$$

$$\frac{\begin{array}{c}\{\mathrm{K} \vdash A_i :: \kappa\}^{i \in 1..r} \qquad \{\mathrm{K} \vdash t_j :: \kappa_j\}^{j \in 1..s} \\ \varphi \equiv \{(a_i \mapsto A_i)\}^{i \in 1..r}_{\cdot} \; \{\cdot (b_j \mapsto t_j)\}^{j \in 1..s}\end{array}}{\mathrm{K} \vdash^{pat} \left(\{A_i\}^{i \in 1..r}_{,} \mid \{t_j\}^{j \in 1..s}_{,}\right) :: \left(\{a_i :: \kappa\}^{i \in 1..r}_{,} \mid \{b_j :: \kappa_j\}^{j \in 1..s}_{,}\right) \rightsquigarrow \varphi} \quad \text{(tat)}$$

Figure 9.5: Kind checking of type and type argument tuples

We require the arities of the tuple and the tuple pattern to match, and the types (or type arguments) must be of the kinds specified in the pattern. Note that rules are more general than allowed by the current syntax in that the generic variables are checked against some kind κ not restricted to $*$. This is because we will extend the syntax of type tuples in Chapter 12 on generic abstraction.

Two type tuples can be compared; the corresponding judgment is dependent on a kind environment K and reads

$$\mathrm{K} \vdash \vartheta_1 \leqslant \vartheta_2 \ .$$

The rule is shown in Figure 9.6. A simple example for an application of comparison is the judgment

$$a_1 :: *, a_2 :: * \vdash a_1, a_2 \leqslant a_1, a_1 \ .$$

We can choose

$$\begin{aligned} \pi &\equiv (b_1 :: *, b_2 :: *) \\ \varphi_1 &\equiv (b_1 \mapsto a_1) \cdot (b_2 \mapsto a_1) \end{aligned}$$

$$\mathrm{K} \vdash \vartheta_1 \leqslant \vartheta_2$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash^{pat} \vartheta_1 :: \pi \rightsquigarrow \varphi_1 \\ \mathrm{K} \vdash^{pat} \vartheta_2 :: \pi \rightsquigarrow \varphi_2 \\ \psi \cdot \varphi_1 \equiv \varphi_2 \\ \mathrm{K} \vdash \psi \text{ kind preserving}\end{array}}{\mathrm{K} \vdash \vartheta_1 \leqslant \vartheta_2} \quad \text{(tt-compare)}$$

Figure 9.6: Comparison of type tuples

$$\vdash \vartheta \downarrow$$

$$\frac{}{\vdash \{a_i\}_{,}^{i \in 1..r} \mid \{t_j\}_{,}^{i \in 1..s} \downarrow} \quad \text{(tt-bounded)}$$

Figure 9.7: Bounded type tuples

$$\begin{aligned} \varphi_2 &\equiv (b_1 \mapsto a_1) \cdot (b_2 \mapsto a_2) \\ \psi &\equiv (a_2 \mapsto a_1) \end{aligned}$$

to show that (a_1, a_2) is indeed smaller than (a_1, a_1). Here are some more examples:

$$\begin{array}{lll} a_1 :: *, a_2 :: *, b_1 :: * & \vdash a_1, a_2 \mid b_1 & \leqslant a_1, a_2 \mid \mathrm{Int} \\ a_1 :: *, a_2 :: *, b_1 :: * & \vdash a_1, a_2 \mid b_1 & \leqslant a_2, a_1 \mid [b_1] \\ a_1 :: * & \vdash a_1 & \leqslant \mathrm{Char} \end{array} .$$

We will make use of the ordering relation on type tuples throughout the following sections.

Furthermore, we say that a type tuple is **bounded**, and write

$$\vdash \vartheta \downarrow ,$$

if all generic slots of the type tuple ϑ consist of type variables. The corresponding rule is defined in Figure 9.7.

9.6 Multi-argument type signatures

Type signatures of type-indexed functions are now of the form

$$x \ \langle \pi \rangle :: (\{y_k \ \langle \vartheta_k \rangle\}^{k \in 1..n}) \Rightarrow t \ .$$

The variables that occur in the pattern π are supposed to be bound in the ϑ_k and in t (thus, alpha-conversion is possible). If π is a $\langle r \mid s \rangle$-ary type tuple pattern, then x is said to be of **arity** $\langle r \mid s \rangle$.

Two things have changed compared to the situation of Chapter 6: the type signature has no longer just one argument, but possibly many; the associated kind is of the same form as that of a type tuple. And the dependencies y_k are now associated with type tuples ϑ_k, which specify in what way x depends on y_k. This is a generalization of the former setting: previously, all type-indexed functions were of arity $\langle 1 \mid 0 \rangle$. As a consequence of that, there was no need for the type tuples to go with the dependencies.

In this and the next section, we will revise the theory of Section 6.3 to work with multi-argument type signatures. Most of the judgments and rules introduced there are replaced with more general variants. We call the generalized language FCR+tif+mpar instead of FCR+tif+par.

Type signatures

$\sigma \quad ::= (\{y_k \ \langle \vartheta_k \rangle\}_{,}^{k \in 1..n}) \Rightarrow t \quad$ type signature of type-indexed function

Figure 9.8: Generalized type signatures in FCR+tif+mpar, replaces type signatures from Figure 6.1

We still use σ to abbreviate the right hand side of a type signature, the part that is of the form $(\{y_k \ \langle \vartheta_k \rangle\}^{k \in 1..n}) \Rightarrow t$ (cf. Figure 9.8).

We also retain the notion of a *base type*. It is no longer instantiated to a type argument, but to a type argument tuple. The revised judgment is shown in Figure 9.9. If the base type of x is instantiated to Θ, then a type signature for x must be in Γ. The type argument Θ must be kind compatible with the pattern π in the type signature. The returned type is the type of the type signature without dependencies, and the variables from the pattern π substituted by the types in Θ, via substitution φ.

We still use the function dependencies from Figure 6.6 to access the dependencies of a type-indexed function. In addition, we need a function deptt that determines in which way one function depends on another. We need to access the information that is encoded in the type tuple associated with the dependency in the type signature. An invocation of deptt takes the form

$$\mathsf{mbase}_{K;\Gamma}(x\ \langle\Theta\rangle) \equiv t$$

$$\frac{\begin{array}{c} x\ \langle\pi\rangle :: (\{y_k\ \langle\vartheta_k\rangle\}_{,}^{k\in 1..n}) \Rightarrow t \in \Gamma \\ K \vdash^{pat} \Theta :: \pi \rightsquigarrow \varphi \end{array}}{\mathsf{mbase}_{K;\Gamma}(x\ \langle\Theta\rangle) \equiv \varphi\ t} \quad \text{(base-m)}$$

Figure 9.9: Revised base type judgment, replaces rule (base) from Figure 6.6

$$\mathsf{deptt}_{K;\Gamma}(x\ \langle\Theta_1\rangle, y) \equiv \Theta_2\ .$$

This means that under environments K and Γ, if function x is instantiated to type argument tuple Θ_1, and it depends on function y, then it will do so with type argument tuple Θ_2. Figure 9.6 shows how to compute this function. A type

$$\mathsf{deptt}_{K;\Gamma}(x\ \langle\Theta_1\rangle, y) \equiv \Theta_2$$

$$\frac{\begin{array}{c} x\ \langle\pi\rangle :: (\{y_k\ \langle\vartheta_k\rangle\}^{k\in 1..n}) \Rightarrow t \in \Gamma \\ h \in 1\,..\,n \\ K \vdash^{pat} \Theta_1 :: \pi \rightsquigarrow \varphi \end{array}}{\mathsf{deptt}_{K;\Gamma}(x\ \langle\Theta_1\rangle, y_h) \equiv \varphi\ \vartheta_h} \quad \text{(deptt)}$$

Figure 9.10: Dependency judgment

signature for x has to be in the environment, and the function function y_h must indeed be one of the dependencies of x. Therefore, there is an associated type tuple ϑ_h. The type argument tuple Θ that x is instantiated to must match the type tuple pattern π in the type signature of x. The match results in a substitution φ for the variables in π, that may also occur in ϑ_h. Therefore, we apply φ to ϑ_h and return the result.

The type signatures of the type-indexed functions that we have introduced so far fit into this scheme easily. For instance, for *map* we have

$$map\ \langle a_1 :: *, a_2 :: *\rangle :: (map\ \langle a_1, a_2\rangle) \Rightarrow a_1 \to a_2$$

and can state that

$$\begin{aligned}
&\mathsf{mbase}(\mathit{map}\ \langle \mathrm{Int}, \mathrm{Char}\rangle) &&\equiv (\mathrm{Int} \to \mathrm{Char})\\
&\mathsf{deptt}(\mathit{map}\ \langle \mathrm{Int}, \mathrm{Char}\rangle, \mathit{map}) &&\equiv (\mathrm{Int}, \mathrm{Char})\ \ .
\end{aligned}$$

Similarly, for *collect* we have

$$\mathit{collect}\ \langle a :: * \mid b :: * \rangle :: (\mathit{collect}\ \langle a \mid b \rangle) \Rightarrow a \to [c]$$

and can derive that

$$\begin{aligned}
&\mathsf{mbase}(\mathit{collect}\ \langle \mathrm{Tree\ Int} \mid \mathrm{Int}\rangle) &&\equiv (\mathrm{Tree\ Int} \to [\mathrm{Int}])\\
&\mathsf{deptt}(\mathit{collect}\ \langle \mathrm{Tree\ Int} \mid \mathrm{Int}\rangle, \mathit{collect}) &&\equiv (\mathrm{Tree\ Int} \mid \mathrm{Int})\ \ .
\end{aligned}$$

For *check*, which we had introduced with type signature

$$\mathit{check}\ \langle a :: * \rangle :: (\mathit{map}\ \langle a, a\rangle, \mathit{collect}\ \langle a \mid \mathrm{Int}\rangle) \Rightarrow a \to \mathrm{Bool}\ ,$$

we can apply the deptt judgment to get

$$\begin{aligned}
&\mathsf{deptt}(\mathit{check}\ \langle \mathrm{Float}\rangle, \mathit{map}) &&\equiv (\mathrm{Float}, \mathrm{Float})\\
&\mathsf{deptt}(\mathit{check}\ \langle \mathrm{Float}\rangle, \mathit{collect}) &&\equiv (\mathrm{Float} \mid \mathrm{Int})\ \ .
\end{aligned}$$

$$\mathrm{K}; \Gamma \vdash^{\mathit{tpsig}} x\ \langle \pi \rangle :: \sigma$$

$$\frac{\begin{gathered}
\{\mathrm{K}; \Gamma \vdash^{\mathit{tpsig}} y_k\ \langle \pi_k \rangle :: (\{z_{k,i}\ \langle \vartheta_{k,i} \rangle\}_{,}^{i \in 1..m_k}) \Rightarrow t_k\}^{k \in 1..n}\\
\{\{z_{k,i} \in \{y_j\}_{,}^{j \in 1..n}\}^{i \in 1..m_k}\}^{k \in 1..n}\\
\mathrm{K}' \equiv \mathrm{K}\ \{, a_i :: *\}^{i \in 1..r}\ \{, b_j :: \kappa_j\}^{j \in 1..s}\\
\Big\{\big\{\{z_{k,i} \equiv y_j \Longrightarrow \mathrm{K}', \pi_k \vdash \vartheta_{k,i} \leqslant \vartheta_k\}^{j \in 1..n}\big\}^{i \in 1..m_k}\Big\}^{k \in 1..n}\\
\mathrm{K}' \vdash t :: * \qquad \{\mathrm{K}' \vdash \vartheta_k :: \pi_k\}^{k \in 1..n} \qquad \{\vdash \vartheta_k \downarrow\}^{k \in 1..n}
\end{gathered}}{\begin{gathered}\mathrm{K}; \Gamma \vdash^{\mathit{tpsig}} x\ \big\langle \{a_i :: *\}_{,}^{i \in 1..r} \mid \{b_j :: \kappa_j\}_{,}^{j \in 1..s} \big\rangle\\
:: \big(\{y_k\ \langle \vartheta_k \rangle\}_{,}^{k \in 1..n}\big) \Rightarrow t\big)\end{gathered}} \quad \text{(typesig-m)}$$

Figure 9.11: Revised well-formedness of type signatures for type-indexed functions, replaces rule (typesig) of Figure 6.7

The correctness of type signatures is a bit harder to verify in this generalized setting. The old rule (typesig) in Figure 6.7 checked the kind correctness of the type and the transitivity of the direct dependency relation. In addition, we now have to verify that the way in which one function depends on another is legal.

The new rule (typesig-m) is given in Figure 9.11. All dependencies y_k of x must have a valid type signature in scope. All indirect dependencies $z_{k,i}$ must be direct dependencies.

There are two new conditions: the type tuple that is associated with an indirect dependency has to be smaller than the type tuple associated with a direct dependency (we use π_k here to extend the kind environment, in the obvious meaning); and each type tuple that is associated with a direct dependency has to be bounded.

The remaining conditions of rule (typesig-m) deal with kind correctness: environment K' is K extended with the bindings from the type tuple pattern that is associated with x. Each type tuple ϑ_k that appears with a dependency y_k of x must match the pattern π_k that is associated with the type signature of y_k. Furthermore, the base type t must be of kind $*$.

Both new conditions are a consequence of how calls to type-indexed functions are translated. Here is an example for why we need the boundedness: consider type-indexed functions x and y with type signatures

$$\begin{array}{l} x\ \langle a\rangle :: (y\ \langle \mathrm{Char}\rangle) \Rightarrow a \\ y\ \langle a\rangle :: (y\ \langle a\rangle) \Rightarrow a\ \ . \end{array}$$

The type tuple Char that is associated with y in x violates the boundedness condition. The generic application $x\ \langle[\mathrm{Int}]\rangle$ is translated to

$$\mathtt{cp}(x,[\,])\ \mathtt{cp}(y,\mathrm{Int})\ ,$$

applying the generic translation algorithm from Figure 6.18 in Section 6.4.7. However, using the generalized generic application algorithm that will be introduced in the following Section 9.7, we can conclude that the FCR type of $\mathtt{cp}(x,[\,])$ would be $\forall a :: *.\ \mathrm{Char} \to [a]$, because the dependency on y is fixed to type Char. But $\mathtt{cp}(y,\mathrm{Int})$ is of type Int, and this is type incorrect.

To see why the absence of the ordering condition between direct and indirect dependencies causes problems, consider the three type-indexed functions x, y, and z with type signatures

$$\begin{array}{lll} x\ \langle a_1,a_2\rangle & :: (y\ \langle a_1\rangle, z\ \langle a_1,a_2\rangle) & \Rightarrow \mathrm{Int} \\ y\ \langle a\rangle & :: (z\ \langle a,a\rangle) & \Rightarrow \mathrm{Int} \\ z\ \langle a_1,a_2\rangle & :: (z\ \langle a_1,a_2\rangle) & \Rightarrow (a_1,a_2)\ \ . \end{array}$$

The function x depends both directly and indirectly on z. However, the condition that $\vdash a,a \leqslant a_1,a_2$ does not hold here. Now, let us have a look at the fragment

$$\begin{array}{l} \mathbf{let}\ y\ \langle\alpha\rangle = 0 \\ \qquad z\ \langle\alpha\rangle = (1,\texttt{"foo"}) \\ \mathbf{in}\ \ x\ \langle[[\alpha]]\rangle\ \ . \end{array}$$

From the revised generic application algorithm that will be introduced in the next section, we can conclude that the generic application $x \langle [[\alpha]] \rangle$ is of the qualified type

$$\forall (a_1 :: *)\ (a_2 :: *).\ (y \langle \alpha \rangle :: \mathrm{Int}, z \langle \alpha \rangle :: (a_1, a_2)) \Rightarrow \mathrm{Int}$$

The dependencies that arise from the use of α in the generic application of x are defined via local redefinition, so that the entire fragment is of type Int. Thereby, the universally quantified variables a_1 and a_2 are instantiated to Int and [Char]. The fragment translates to

$$\begin{array}{l}
\textbf{let}\ \mathsf{cp}(y,\alpha) = 0 \\
\quad\ \ \mathsf{cp}(z,\alpha) = (1, \texttt{"foo"}) \\
\textbf{in}\ \ \mathsf{cp}(x,[\,])\ \big(\mathsf{cp}(y,[\,])\ \mathsf{cp}(z,\alpha)\big) \\
\qquad\qquad\quad \big(\mathsf{cp}(z,[\,])\ \mathsf{cp}(z,\alpha)\big)\ .
\end{array}$$

The term $\mathsf{cp}(z,\alpha)$ is used in the translation twice – but the first use is type incorrect, because of the restricted type in the dependency of y on z. The type of component $\mathsf{cp}(y,[\,])$ is

$$\forall a :: *.\ (a, a) \rightarrow \mathrm{Int}\ ,$$

and we cannot apply that to a tuple of type (Int, [Char]). By insisting that the type tuples associated with indirect dependencies are always smaller than the type tuples associated with direct dependencies, we prevent such type mismatches.

9.7 Revised generic application algorithm

It is now time to look once more at the generic application algorithm. This algorithm does now exist in two versions: one is still parametrized by a single type argument, because that is the way a type-indexed function is called in a program. The other is the actual replacement for the former algorithm, which now works on type argument tuples.

The first, the wrapper, is shown in Figure 9.12. Syntactically, this call is of the same for than the old generic application algorithm in Figure 6.8. The only rule, (ga), looks up the arity $\langle r \mid s \rangle$ of the generic function x in the environment Γ. The type argument A is replicated r times, and the s type variables b_j are universally quantified in the resulting type. The computation is then delegated to the type tuple version of the algorithm, which is called gmapp.

The algorithm gmapp directly corresponds to the original gapp in Figure 6.8, including the additional rule (ga-4) in Figure 6.19 for type arguments of higher

$$\mathsf{gapp}_{K;\Gamma}(x\ \langle A\rangle) \equiv q$$

$$\frac{x\ \langle\{a_i :: *\}_{,}^{i\in 1..r} \mid \{b_j :: \kappa_j\}_{,}^{j\in 1..s}\rangle :: \sigma \in \Gamma}{\mathsf{gapp}_{K;\Gamma}(x\ \langle A\rangle) \equiv \{\forall b_j :: \kappa_j.\}^{j\in 1..s}\ \mathsf{gmapp}_{K;\Gamma}(x\ \langle\{A\}_{,}^{i\in 1..r} \mid \{b_j\}_{,}^{j\in 1..s}\rangle)} \quad \text{(ga)}$$

Figure 9.12: Wrapper for the revised generic application algorithm

kind. The new rules are shown in Figure 9.13 and 9.14. The judgments are of the form

$$\mathsf{gmapp}_{K;\Gamma}(x\ \langle\Theta\rangle) \equiv q\ ,$$

the only difference except for the name being that we parametrize over a type argument tuple Θ now. To create a single dependency constraint, we now use the auxiliary judgment

$$\mathsf{mkmdep}_{K;\Gamma}(y\ \langle\alpha \leftarrow \Theta\rangle) \equiv Y\ ,$$

again with a type argument tuple in place of the former single type variable.

In the original algorithm, we distinguish cases based on the dependency variables that occur in the type argument. We still do the same thing, based on the dependency variables that occur in the type argument tuple. The generic slots in the type argument tuple are always of the same shape. They are all of the same kind, and do all contain the same dependency variables in the same positions. This uniformity is easy to see: the wrapper gapp places a single type argument A in all generic slots of the type argument tuple, and during the whole algorithm we only perform operations on the type argument that preserve uniformity, such as substituting one dependency variable in all generic slots. The non-generic slots of the type argument tuple are never really involved in the algorithm and passed around everywhere without modification. Only the wrapper gapp performs universal quantification on those, on the outside of the resulting qualified type.

It thus makes sense to distinguish the following four cases (they correspond one by one to the four cases of the original algorithm): rule (ga-m-1) covers type argument tuples where no dependency variables appear and where the generic slots are all of kind $*$ – we use the judgment $K \vdash \Theta :: *$ to denote this particular property.

The second case (ga-m-2) is for type argument tuples in which the dependency variable α forms the head of all generic slots. The third case (ga-m-3) is for type

$$\mathsf{gmapp}_{K;\Gamma}(x\ \langle\Theta\rangle) \equiv q$$

$$\frac{K \vdash \Theta :: * \qquad \mathsf{fdv}(\Theta) \equiv \varepsilon}{\mathsf{gmapp}_{K;\Gamma}(x\ \langle\Theta\rangle) \equiv \mathsf{mbase}(x\ \langle\Theta\rangle)} \quad \text{(ga-m-1)}$$

$$\frac{K \vdash \alpha :: \kappa \qquad \{a_i \text{ fresh}\}^{i\in 1..r} \qquad K' \equiv K\ \{,a_i :: \kappa\}^{i\in 1..r}}{\begin{array}{l}\mathsf{gmapp}_{K;\Gamma}(x\ \langle\{\alpha\ \{A_{i,h}\}^{h\in 1..\ell_i}\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s}\rangle) \\ \quad \equiv \{\forall a_i :: \kappa.\}^{i\in 1..r}\ \Big(\mathsf{mkmdep}_{K';\Gamma}(y_k\ \langle\alpha \leftarrow \{a_i\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s}\rangle)\Big) \\ \qquad \Rightarrow \mathsf{gmapp}_{K';\Gamma}(x\ \langle\{a_i\ \{A_{i,h}\}^{h\in 1..\ell_i}\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s}\rangle)\end{array}} \quad \text{(ga-m-2)}$$

$$\frac{\begin{array}{c}\alpha \in \mathsf{fdv}(\Theta) \qquad \Theta \equiv \{A_i\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s} \qquad \{\mathsf{head}(A_i) \not\equiv \alpha\}^{i\in 1..r} \\ K \vdash \alpha :: \kappa \qquad \{a_i \text{ fresh}\}^{i\in 1..r} \qquad K' \equiv K\ \{,a_i :: \kappa\}^{i\in 1..r} \\ \mathsf{dependencies}_{\Gamma}(x) \equiv \{y_k\}_{,}^{k\in 1..n} \\ \{\Theta_k \equiv \mathsf{deptt}_{K';\Gamma}(x\ \langle\{a_i\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s}\rangle, y_k)\}^{k\in 1..n}\end{array}}{\begin{array}{l}\mathsf{gmapp}_{K;\Gamma}(x\ \langle\Theta\rangle) \\ \quad \equiv \{\forall a_i :: \kappa.\}^{i\in 1..r}\ (\{\mathsf{mkmdep}_{K';\Gamma}(y_k\ \langle\alpha \leftarrow \Theta_k\rangle)\}_{,}^{k\in 1..n}) \\ \qquad \Rightarrow \mathsf{gmapp}_{K';\Gamma}(x\ \langle\{A_i[\alpha_i\ /\ a_i]\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s}\rangle)\end{array}} \quad \text{(ga-m-3)}$$

$$\frac{\begin{array}{c}K \vdash \Theta :: \kappa \to \kappa' \qquad \mathsf{fdv}(\Theta) \equiv \varepsilon \qquad \Theta \equiv \{A_i\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s} \\ \{a_i \text{ fresh}\}^{i\in 1..r} \qquad K' \equiv K\ \{,a_i :: \kappa\}^{i\in 1..r} \\ \mathsf{dependencies}_{\Gamma}(x) \equiv \{y_k\}_{,}^{k\in 1..n} \\ \{\Theta_k \equiv \mathsf{deptt}_{K';\Gamma}(x\ \langle\{a_i\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s}\rangle, y_k)\}^{k\in 1..n}\end{array}}{\begin{array}{l}\mathsf{gmapp}_{K;\Gamma}(x\ \langle\Theta\rangle) \\ \quad \equiv \{\forall a_i :: \kappa.\}^{i\in 1..r}\ \{\mathsf{gmapp}_{K';\Gamma}(y_k\ \langle\Theta_k\rangle) \to\}^{k\in 1..n} \\ \qquad\qquad \mathsf{gmapp}_{K';\Gamma}(x\ \langle\{A_i\ a_i\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s}\rangle)\end{array}} \quad \text{(ga-m-4)}$$

Figure 9.13: Revised generic application algorithm, replaces Figures 6.8 and 6.19

$$\mathsf{mkmdep}_{K;\Gamma}(y\ \langle \alpha \leftarrow \Theta\rangle) \equiv Y$$

$$\frac{\begin{array}{cc} K \vdash \alpha :: \kappa & \kappa \equiv \{\kappa_h \rightarrow\}^{h\in 1..\ell} * \\ \{\gamma_h \text{ fresh}\}^{h\in 1..\ell} & K' \equiv K\ \{, \gamma_h :: \kappa_h\}^{h\in 1..\ell} \end{array}}{\begin{array}{l} \mathsf{mkmdep}_{K;\Gamma}\bigl(x\ \langle \alpha \leftarrow \{A_i\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s}\rangle\bigr) \\ \quad \equiv \Bigl(x\ \langle \alpha\ \{\gamma_h\}^{h\in 1..\ell}\rangle \\ \qquad\qquad :: \mathsf{gmapp}_{K;\Gamma}\bigl(x\ \langle \{A_i\ \{\gamma_h\}^{h\in 1..\ell}\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s}\rangle\bigr)\Bigr) \end{array}} \quad \text{(mkdep-m)}$$

Figure 9.14: Revised generic application algorithm, continued from Figure 9.13, replaces Figures 6.8 and 6.19

argument tuples where a dependency variable appears in all generic slots, but not in the head.

The fourth rule (ga-m-4) remains for type argument tuples which are without dependency variables, but of functional kind. Again, we use the judgment $K \vdash \Theta :: \kappa \rightarrow \kappa'$ to state that all generic slots of Θ are of kind $\kappa \rightarrow \kappa'$.

The implementation of each of the cases is also only a generalization and thus a more complicated version of the corresponding cases in the original algorithm. In the first case, we instantiate the base type, now via mbase.

In the second case, we create a dependency for the function x on α. Instead of one type variable, we create r fresh type variables a_i of the same kind as α, where r is the number of generic slots in the type argument tuple. When generating the dependency with help of the mkmdep function, we instantiate α with these fresh type variables a_i, but pass on the non-generic types t_j without modification. We then recursively call gmapp with α substituted by the a_i.

The third case generates dependencies for dependency variable α and all functions y_k that x depends on. As in the second case, we create r fresh type variables a_i of the same kind as α, where $\langle r \mid s\rangle$ is the arity of the type argument tuple Θ. We have to respect the way in which x depends on each of the y_k, therefore we pass the type argument tuple

$$\{a_i\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s}\ ,$$

where the original generic entries have been replaced by the fresh type variables, to the deptt function. As a result, we get modified type argument tuples Θ_k which we can use to generate the dependency constraints. In the recursive call, we substitute a_i for α in the i-th generic slot of Θ.

The fourth case is for the situation that all dependency variables have been eliminated, but the generic slots in the type argument tuple Θ are not of kind $*$, but of functional kind $\kappa \to \kappa'$. We treat this situation similarly to the previous case, only that we do not introduce dependency constraints, but explicit function arguments.

It remains to discuss the mkmdep function, which is displayed in Figure 9.14. The call

$$\mathsf{mkmdep}_{\mathrm{K};\Gamma}(x \ \langle y \leftarrow \Theta \rangle) \equiv Y$$

expresses that under environments K and Γ, the dependency constraint Y belongs to the dependency y of x for α, if α is instantiated by the type argument tuple Θ.

To compute the constraint, we introduce new local dependency variables γ_h, guided by the arity of the kind κ of α. Note that the generic slots of the type argument tuple Θ are of the same kind κ. Therefore, the γ_h are used as arguments not only for α in the name of the dependency constraint that is generated, but also as arguments for the generic slots A_i of Θ in the recursive call to gmapp.

9.8 Multiple dependencies on one function

There is a restriction that a dependency constraint set must contain at most one entry per function and dependency variable. Similarly, the type signature of a type-indexed function must contain at most one dependency on a certain other function.

In the situation of Chapter 6, this was not really a limitation, because one type-indexed function could depend on another in only one way. Now, where each dependency is associated with a type tuple, there might be the need to depend on the same function more than once, in different ways.

As an example, consider the following implementation of a product case for a hypothetical generic function *combine* which takes two values and performs some analysis whether they can be combined:

$$\mathit{combinable}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle\ (x_1 \times x_2)\ (y_1 \times y_2) = \mathit{special}\ \langle \alpha \rangle\ x_1 \vee \mathit{special}\ \langle \alpha \rangle\ y_1 \ .$$

Two pairs can be combined if one of the first components is special in the sense that another generic function *special* returns *True* on the value.

If we assume that *combinable* has the type signature

$$\begin{aligned} \mathit{combinable}\ \langle a :: * \rangle \quad & :: (\mathit{combinable}\ \langle a \rangle, \mathit{special}\ \langle a \rangle) \\ & \quad \Rightarrow a \to a \to \mathrm{Bool}\ , \end{aligned}$$

then everything is fine. However, if we want the possibility to combine values of different types, we need two dependencies on *special*:

$$\begin{aligned} &\mathit{combinable}\ \langle a_1 :: *, a_2 :: * \rangle :: (\mathit{combinable}\ \langle a_1, a_2\rangle, \mathit{special}\ \langle a_1\rangle, \mathit{special}\ \langle a_2\rangle) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow a_1 \to a_2 \to \mathrm{Bool}\ , \end{aligned}$$

and the two calls to *special* $\langle \alpha \rangle$ above would refer to different dependencies. We do not allow such double dependencies.

Instead, we enable multiple dependencies to the same function via an easy way to create copies of an existing type-indexed function. This is achieved via *default cases*, which are introduced in Chapter 14 and are mainly used to create variants of existing type-indexed functions by *extending* them. The facility to create copies of functions is a special case that is useful here: we can write

special′ **extends** *special*

to create a copy of *special* and then use the legal type signature

$$\begin{aligned} &\mathit{combinable}\ \langle a_1 :: *, a_2 :: * \rangle :: (\mathit{combinable}\ \langle a_1, a_2\rangle, \mathit{special}\ \langle a_1\rangle, \mathit{special}'\ \langle a_2\rangle) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow a_1 \to a_2 \to \mathrm{Bool}\ , \end{aligned}$$

with the definition

$$\mathit{combinable}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle\ (x_1 \times x_2)\ (y_1 \times y_2) = \mathit{special}\ \langle \alpha \rangle\ x_1 \vee \mathit{special}'\ \langle \alpha \rangle\ y_1\ ,$$

where *special′* and *special* implement the same function.

9.9 Translation and correctness

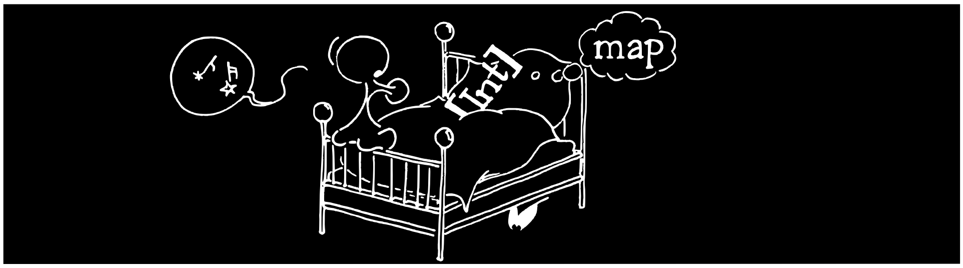
In the translation of FCR+tif+par to FCR that we have introduced in Chapter 6, we refer to the original generic application algorithm gapp. The only change that we need in order to obtain a translation of FCR+tif+mpar to FCR is to interpret each call to the original gapp algorithm as a call to the new gapp wrapper. Everything else works exactly as before.

The effect of the change is that in some situations, we allow more liberal types for generic applications or arms of type-indexed functions. We thus have to verify the correctness of the translation once more, to ensure that the translation actually warrants the more liberal types we assign. We require a variant of Theorem 6.5 that is adapted to type argument tuples.

Theorem 9.1 (Correctness of revised generic application). If we have are in the situation that $\mathsf{gmapp}_{\mathrm{K};\Gamma}(x\ \langle \Theta \rangle) \equiv q$ and $\mathrm{K}; \Delta \vdash q \leqslant t$, then

$$[\![\mathrm{K}; \Gamma; \Delta; \Sigma]\!]^{\mathrm{par}}_{\mathrm{K};\Gamma} \vdash [\![x\ \langle \Theta \rangle]\!]^{\mathrm{gtrans}}_{\mathrm{K};\Gamma;\Sigma} :: t\ .$$

10 Embedding Datatypes



In order to specialize a generic function to a type that is not in the signature of that function (i.e., for which the function is not explicitly defined), it is helpful to realize that we can map each datatype to an isomorphic *structural representation type*. In Section 7.5.1, we have sketched that such a structural representation replaces the toplevel structure of a datatype with occurrences of the datatypes Unit, Sum, and Prod (all introduced in Section 7.1), but leaves the fields and possible recursive calls intact.

The details of the translation are not as essential as the fact that the translation is an embedding of all datatypes into type expressions, making use of a limited number of additional datatypes. Using this embedding, we can reduce the problem of specializing a generic function to a new datatype to the following two subproblems: first, we can specialize the generic function to the structural representation of the datatype instead. Second, we need to find a way to exploit the fact that the structural representation is isomorphic to the original type in such a way that we can turn the specialization on the representation type into a function that works for the original type.

In this chapter, we formally define the embedding of datatypes into the type language, and we will convince ourselves that the structural representations are really isomorphic to the original types. In the Chapter 11, we will focus on how to exploit the structural representation for the specialization of calls to generic functions.

10.1 Zero

A close look at the grammar of FC in Figure 3.1 reveals that a datatype is allowed to have no constructors at all. The declaration

data Zero =

is valid in FC (and thus, in all languages based upon it). In Haskell 98, this is not a legal declaration, but datatypes without constructors are allowed by some implementations as an extension – including the GHC, where one can simply write

data Zero

to define such a type.

There is no way to construct a value of the type Zero, therefore Zero is a type without any values (except for the undefined value $\perp$). Nevertheless, the type is sometimes useful, particularly in combination with other datatypes. For parametrized types, Zero can be used to exclude certain alternatives – for instance, the type [Zero] of lists of elements of type Zero has (ignoring $\perp$ values or combinations therewith) only one value, the empty list [ ]. We will see a useful application of Zero in the Chapter 16 on type-indexed types.

Unfortunately, the structure of a type with no constructors cannot be represented as a type expression involving Unit, Sum, and Prod. We therefore choose Zero as a fourth special type. Types with no constructors are represented using Zero. Note that there are also parametrized data types that have no constructors. For

data ParZero $(f :: * \to *)$ $(a :: *)$ = ,

the representation type is

type Str(ParZero) $(f :: * \to *)$ $(a :: *)$ = Zero .

Does that mean that we now have to add a case for Zero to every generic definition? Yes and no. Types without values are used very infrequently, and as with any other type, we do not need a case for it as long as we do not want to use the function on such types. For example, if we omit the Zero case for the generic equality function, everything is fine – unless we actually have calls such as *equal* ⟨Zero⟩ or *equal* ⟨[Zero]⟩ in our program, which cause specialization errors.

On the other hand, a library writer might want to add a case for Zero to make the generic functions fit for use also in such rare scenarios.

An example for a function that can actually benefit from an arm for the Zero type is the generic enumeration function *enum* (defined on page 111). We could define *enum* on Zero as follows:

$$\mathit{enum}\ \langle \text{Zero} \rangle = [\,]\ .$$

The *enum* function lists all (non-$\perp$) values of a datatype, and as such computes a characteristic of the type itself rather than performing some operation on values of the type.

10.2 A few examples

The translation to structural representation types is simple enough. Nevertheless, we have only seen one example so far, back in Section 7.5.1, and it can do no harm to discuss the embedding for a few more datatypes.

In Section 7.2, we stated that the type Bool is, unlike Int or Char or Float, not considered abstract, but assumed to be defined as

$$\mathbf{data}\ \text{Bool} \quad = \mathit{False} \mid \mathit{True}\ .$$

Indeed, it is easy to give the structural representation of Bool,

$$\mathbf{type}\ \mathsf{Str}(\text{Bool}) = \text{Sum Unit Unit}\ ,$$

and the associated embedding-projection pair is trivial.

Likewise, the type of lists, although we assume special syntax for it, is not considered abstract, but assumed to be defined as

$$\mathbf{data}\ [a :: *] \quad = [\,] \mid a : [a]\ .$$

If we translate the special syntax away and assume the type constructor to be List, the "nil" constructor to be *Nil*, and the "cons" constructor to be *Cons*, this would correspond to the definition

$$\mathbf{data}\ \text{List}\ (a :: *) = \mathit{Nil} \mid \mathit{Cons}\ a\ (\text{List}\ a)\ .$$

The structural representation of lists is

$$\mathbf{type}\ \mathsf{Str}([\,])\ (a :: *) = \text{Sum Unit (Prod}\ a\ [a])\ ,$$

and the associated embedding-projection pair is:

$$\begin{aligned}
\mathsf{ep}([\,]) = \textbf{let}\; & \mathit{fromList}\; [\,] && = \mathit{Inl}\; \mathit{Unit} \\
& \mathit{fromList}\; (x : xs) && = \mathit{Inr}\; (x \times xs) \\
& \mathit{toList}\quad (\mathit{Inl}\; \mathit{Unit}) && = [\,] \\
& \mathit{toList}\quad (\mathit{Inr}\; (x \times xs)) && = (x : xs) \\
\textbf{in}\;\; & \mathit{EP}\; \mathit{fromList}\; \mathit{toList}\;\; .
\end{aligned}$$

Look at the definitions of *fromList* and *toList*. The patterns in *fromList* appear as right hand sides of *toList*, and the patterns of *toList* double as right hand sides of *fromList*. This symmetry holds in general, and makes it obvious that *fromList* and *toList* are indeed inverses of each other, and thus define isomorphisms. Note that the structural representation of List cannot be distinguished from the representation of $[\,]$, because the representation types do not (yet) contain any information about the names of the constructors or the original datatype.

Structural representations are never directly recursive. The fields in the original datatype are not touched by the translation, and as a consequence, $\mathsf{Str}([\,])$ still refers to the original type $[\,]$. The translation replaces the top layer of a datatype with its structure. If necessary, the translation can be applied repeatedly, to reveal the structure of the fields as well.

If a datatype has only one constructor, then no application of Sum occurs in the structural representation. An example is the datatype of rose trees, defined as

$$\textbf{data}\ \mathrm{Rose} \qquad (a :: *) = \mathit{Fork}\; a\; [\mathrm{Rose}\; a] \;\; .$$

A rose tree is a tree with a variable degree of branching. In each node, it stores a value and a list of possible subtrees. There is no need for a separate constructor to represent a leaf, we can simply use the auxiliary function

$$\mathit{roseleaf}\; x = \mathit{Fork}\; x\; [\,]$$

to construct a leaf. The representation type for rose trees is

$$\textbf{type}\ \mathsf{Str}(\mathrm{Rose})\ (a :: *) = \mathrm{Prod}\; a\; [\mathrm{Rose}\; a] \;,$$

the associated embedding-projection pair is

$$\begin{aligned}
\mathsf{ep}(\mathrm{Rose}) = \textbf{let}\; & \mathit{fromRose}\; (\mathit{Fork}\; x\; y) && = x \times y \\
& \mathit{toRose}\quad (x \times y) && = \mathit{Fork}\; x\; y \\
\textbf{in}\;\; & \mathit{EP}\; \mathit{fromRose}\; \mathit{toRose}
\end{aligned}$$

If a datatype has more than two constructors, or more than two fields in a constructor, we use nested applications of Sum and Prod to represent the datatype.

We have already seen one such example, Tree (where there are three fields in the *Node* constructor), in Section 7.5.1. Another one is a datatype such as Direction, defined as

```
data Direction        = North | East | South | West ,
```

where the structural representation and embedding-projection pair are defined as follows:

```
type Str(Direction) = Sum Unit (Sum Unit (Sum Unit Unit))
ep(Direction) =
  let fromDirection North              = Inl Unit
      fromDirection East               = Inr (Inl Unit)
      fromDirection South              = Inr (Inr (Inl Unit))
      fromDirection West               = Inr (Inr (Inr Unit))
      toDirection   (Inl Unit)         = North
      toDirection   (Inr (Inl Unit))   = East
      toDirection   (Inr (Inr (Inl Unit))) = South
      toDirection   (Inr (Inr (Inr Unit))) = West
  in  EP fromDirection toDirection .
```

Note that we have chosen a biased encoding for the sum structure, nesting to the right, and we do the same for the product structure. We will discuss other choices briefly in Chapter 17.

To summarize: for datatypes without constructors, Zero is used as representation; if there is one constructor, no sum structure is generated, as in the Rose example; if there are two or more constructors, we produce a nested sum. For each constructor, we proceed similarly: if there are no fields, we use Unit to represent the constructor; one field is represented by itself; for multiple fields, a nested product is created.

In the following section, we will formalize both the generation of structural representations and of embedding-projection pairs.

10.3 Formal translation

Let us now look at the translation in detail. We define two functions, one to construct a parametrized type from the definition of a datatype, and one to build the definition of the embedding-projection pair converting between the original type and its structural representation. A **parametrized type** is of the form

$$\{\Lambda a_i :: \kappa_i .\}^{i \in 1..\ell}\ t\ ,$$

a type t with a number of kind-annotated type-level lambda abstractions in front. The a_i have to be all different and are considered to be bound in t. We extend kind checking to parametrized types using the new rule in Figure 10.1.

$$\mathrm{K} \vdash t :: \kappa$$

$$\frac{\mathrm{K}\ \{, a_i :: \kappa_i\}^{i\in 1..\ell} \vdash t :: \kappa}{\mathrm{K} \vdash \{\Lambda a_i :: \kappa_i.\ t\}^{i\in 1..\ell} :: \{\kappa_i \rightarrow\}^{i\in 1..\ell}\ \kappa} \quad \text{(t-par)}$$

Figure 10.1: Kind checking of parametrized types, extends Figure 3.2

We will use parametrized types to represent the generic representations of datatypes. Before, in the examples, we have used Haskell type synonyms, using the **type** keyword. But **type** is not part of our FCR language. The structural representation types are only used internally, therefore we can use parametrized types that are applied to arguments where needed, instead of extending our language with an additional construct.

The translation to the structural representation is expressed by judgments of the forms

$$\begin{aligned} &[\![D]\!]^{\mathrm{str}} && \equiv \{\Lambda a_i :: \kappa_i.\}^{i\in 1..\ell}\ t \\ &[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} && \equiv t\ . \end{aligned}$$

In the first case, a datatype declaration is mapped to a parametrized type; in the second case, a list of constructors is mapped to a type. The rules are shown in Figures 10.2 and 10.3.

The main part of the work happens during the translation of the constructors. For a complete datatype, handled by rule (str-data), the type variables over which it is parametrized are stripped. Then, the bare constructors are translated. In the end, the variables are reinserted without modification to form the parameters of the structural representation.

For constructors, we distinguish several different cases. In the special case that there is no constructor, rule (str-constr-1) applies, and we translate to Zero, as explained in Section 10.1. The datatype Zero is used only in this case.

The next three rules deal with a single constructor. Again, we distinguish different cases. If the constructor is without fields, the rule (str-constr-2) translates it to Unit. Again, Unit is only used in this specific case. If the constructor has one field t, the translation is the field itself according to rule (str-constr-3). Only if at

$$[\![D]\!]^{\mathrm{str}} \equiv \{\Lambda a_i :: \kappa_i.\}^{i\in 1..\ell}\ t$$

$$\frac{[\![\{(C_j\ \{t_{j,k}\}^{k\in 1..n_j})\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} \equiv t}{[\![\mathbf{data}\ T = \{\Lambda a_i :: \kappa_i.\}^{i\in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} \equiv \{\Lambda a_i :: \kappa_i.\}^{i\in 1..\ell}\ t} \quad \text{(str-data)}$$

Figure 10.2: Structural representation of datatypes

$$[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} \equiv t$$

$$\frac{}{[\![\varepsilon]\!]^{\mathrm{str}} \equiv \mathrm{Zero}} \quad \text{(str-constr-1)} \qquad \frac{}{[\![C]\!]^{\mathrm{str}} \equiv \mathrm{Unit}} \quad \text{(str-constr-2)}$$

$$\frac{}{[\![C\ t]\!]^{\mathrm{str}} \equiv t} \quad \text{(str-constr-3)}$$

$$\frac{n \in 2\,.\,. \qquad [\![C\ \{t_k\}^{k\in 2..n}]\!]^{\mathrm{str}} \equiv t'_2}{[\![C\ \{t_k\}^{k\in 1..n}]\!]^{\mathrm{str}} \equiv \mathrm{Prod}\ t_1\ t'_2} \quad \text{(str-constr-4)}$$

$$\frac{\begin{array}{c} m \in 2\,.\,. \qquad [\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 2..m}]\!]^{\mathrm{str}} \equiv t_2 \\ [\![C_1\ \{t_{1,k}\}^{k\in 1..n_1}]\!]^{\mathrm{str}} \equiv t_1 \end{array}}{[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} \equiv \mathrm{Sum}\ t_1\ t_2} \quad \text{(str-constr-5)}$$

Figure 10.3: Structural representation of constructors

least two fields are present, a product structure is created in rule (str-constr-4). The constructor with all but the first field is recursively translated (but there still is at least one) to t_2', whereas the first field is included in the result Prod t_1 t_2' unchanged.

The final rule, (str-constr-5), is for the situation that there are at least two constructors. Only in this case a sum structure is created, analogous to the product structure. The first constructor is translated on its own, and all remaining constructors (there is still at least one) are translated as a sequence, and the result is Sum applied to the two partial translations.

Theorem 10.1. Assume that K contains the bindings necessary for the embedding, Zero :: $*$, Unit :: $*$, Sum :: $*^2$, and Prod :: $*^2$. If D is a well-formed datatype, thus $\mathrm{K} \vdash D \rightsquigarrow T :: \kappa; \Gamma$, then $\mathrm{K} \vdash [\![D]\!]^{\mathrm{str}} :: \kappa$.

In other words, the structural representation is of the same kind as the original datatype.

Proof. The well-formedness of the datatype declaration (cf. rule (p-data) in Figure 3.6) implies that all constructor fields are of kind $*$ under environment $\mathrm{K}' \equiv \mathrm{K}\ \{a_i :: \kappa_i\}^{i \in 1..\ell}$, where the a_i are the parameters of the datatype. Under this condition, it is easy to see that if any judgment of the form

$$[\![\{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}]\!]^{\mathrm{str}} \equiv t$$

holds, then $\mathrm{K}' \vdash t :: *$. This holds thus for the specific type t that is the result of the constructor translation in rule (str-data). Application of the new kind checking rule (t-par) yields

$$\mathrm{K}' \vdash \{\Lambda a_i :: \kappa_i.\}^{i \in 1..\ell}\ t :: \kappa\ ,$$

which is the claim. □

The structural representation type of a datatype is also called the **standard view** of the datatype, as opposed to alternative views that are the topic of Chapter 17.

The translation that generates embedding-projection pairs consists of judgments of the forms

$$\begin{aligned} &[\![D]\!]^{\mathrm{ep}} && \equiv d \\ &[\![\{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}]\!]^{\mathrm{ep}} && \equiv \{p_{\mathrm{from}\,j}\}_{|}^{j \in 1..m}; \{p_{\mathrm{to}\,j}\}_{|}^{j \in 1..m}\ . \end{aligned}$$

The rules that are presented in Figures 10.4 and 10.5 are exactly analogous to the rules for generating the structural representations. Not only do we distinguish

$$[\![D]\!]^{\mathrm{ep}} \equiv d$$

$$\frac{\begin{array}{c}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{ep}} \equiv \{p_{\mathrm{from}\,j}\}_{|}^{j\in 1..m}; \{p_{\mathrm{to}\,j}\}_{|}^{j\in 1..m} \\ d_{\mathrm{from}} \equiv x_{\mathrm{from}} = \lambda y \to \mathbf{case}\ y\ \mathbf{of}\ \{p_{\mathrm{from}\,j} \to p_{\mathrm{to}\,j}\}_{;}^{j\in 1..m} \\ d_{\mathrm{to}} \equiv x_{\mathrm{to}} = \lambda y \to \mathbf{case}\ y\ \mathbf{of}\ \{p_{\mathrm{to}\,j} \to p_{\mathrm{from}\,j}\}_{;}^{j\in 1..m} \\ e \equiv \mathbf{let}\ \{d_{\mathrm{from}}; d_{\mathrm{to}}\}\ \mathbf{in}\ EP\ x_{\mathrm{from}}\ x_{\mathrm{to}}\end{array}}{[\![\mathbf{data}\ T = \{\Lambda a_i :: \kappa_i.\}^{i\in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{ep}} \equiv \mathtt{ep}(T) = e} \quad \text{(ep-data)}$$

Figure 10.4: Generation of embedding-projection pairs for datatypes

between the translation of a complete datatype declaration D to a value declaration d again and the translation of a list of constructors to a pair of two lists of patterns. Also, the five rules for the translation of constructors correspond one by one to the five rules in Figure 10.3. Structural representation type and embedding-projection pair generation can thus easily be fused in an implementation.

The rule (ep-data) processes the declaration for one datatype T. The declaration produced is for the value of the name $\mathtt{ep}(T)$. We assume that the function $\mathtt{ep}$ maps a type name to a variable name in a way that it does not clash with any user-defined names. The right hand side of the value declaration is of the form

$$\begin{array}{ll}\mathbf{let}\ x_{\mathrm{from}} & = \lambda y \to \mathbf{case}\ y\ \mathbf{of}\ \{p_{\mathrm{from}\,j} \to p_{\mathrm{to}\,j}\}_{;}^{j\in 1..m} \\ \quad\ x_{\mathrm{to}} & = \lambda y \to \mathbf{case}\ y\ \mathbf{of}\ \{p_{\mathrm{to}\,j} \quad \to p_{\mathrm{from}\,j}\}_{;}^{j\in 1..m} \\ \mathbf{in}\ \ EP\ x_{\mathrm{from}}\ x_{\mathrm{to}}\ \ . & \end{array}$$

This structure makes it very obvious that the EP contains two mutual inverses. Note that patterns are used as expressions here. This is possible because the pattern language is a subset of the expression language, and, as we will see, the rules for translating constructors ensure that $p_{\mathrm{from}\,j}$ always contains exactly the same variables as $p_{\mathrm{to}\,j}$. The shape of the expressions also reveals what the patterns should look like. The $p_{\mathrm{from}\,j}$ should match on a value of the j-th constructor of the datatype, whereas $p_{\mathrm{to}\,j}$ should be a suitable product that is injected into the sum structure of the representation type.

To return these patterns is the job of the rules that translate a list of constructors. Rule (ep-constr-1) states that no constructors translate to no patterns. As a result, the two case statements above will be empty. But in such a situation, also the y's above are of a type with no constructors, and thus no defined values – it

$$[\![\{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}]\!]^{\mathrm{ep}} \equiv \{p_{\mathrm{from}\ j}\}_{|}^{j \in 1..m}; \{p_{\mathrm{to}\ j}\}_{|}^{j \in 1..m}$$

$$\frac{}{\varepsilon \rightsquigarrow \varepsilon; \varepsilon}\ \text{(ep-constr-1)} \qquad \frac{}{[\![C]\!]^{\mathrm{ep}} \equiv C; \mathit{Unit}}\ \text{(ep-constr-2)}$$

$$\frac{x \text{ fresh}}{[\![C\ t]\!]^{\mathrm{ep}} \equiv C\ x; x}\ \text{(ep-constr-3)}$$

$$\frac{n \in 2\,.\,. \qquad x_1 \text{ fresh} \qquad [\![C\ \{t_k\}^{k \in 2..n}]\!]^{\mathrm{ep}} \equiv C\ \{x_k\}^{k \in 2..n}; p_{\mathrm{to}}}{[\![C\ \{t_k\}^{k \in 1..n}]\!]^{\mathrm{ep}} \equiv C\ \{x_k\}^{k \in 1..n}; x_1 \times p_{\mathrm{to}}}\ \text{(ep-constr-4)}$$

$$\frac{\begin{array}{c} m \in 2\,.\,. \\ [\![\{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 2..m}]\!]^{\mathrm{ep}} \equiv \{p_{\mathrm{from}\ j}\}_{|}^{j \in 2..m}; \{p_{\mathrm{to}\ j}\}_{|}^{j \in 2..m} \\ [\![C_1\ \{t_{1,k}\}^{k \in 1..n_1}]\!]^{\mathrm{ep}} \equiv p_{\mathrm{from}\ 1}; p_{\mathrm{to}\ 1} \end{array}}{\begin{array}{l} [\![\{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}]\!]^{\mathrm{ep}} \\ \quad \equiv \{p_{\mathrm{from}\ j}\}_{|}^{j \in 1..m}; \mathit{Inl}\ p_{\mathrm{to}\ 1}\ \{|\ \mathit{Inr}\ p_{\mathrm{to}\ j}\}^{j \in 2..m} \end{array}}\ \text{(ep-constr-5)}$$

Figure 10.5: Generation of embedding-projection pairs for constructors

is irrelevant what the conversion function looks like, as it will never be applied to a non-$\perp$ value.

If there is one constructor C without any fields, the constructor is associated with *Unit* in rule (ep-constr-2). If there is one field, as in rule (ep-constr-3), we introduce a new variable to represent the value of that field. Because the representation of a single field is the field itself, we associate the pattern $C\ x$ with x. If there are multiple fields, we introduce a fresh variable x_1 for the first field and map the the constructor to a (possibly nested) product pattern $x_1 \times p_{\text{to}}$, where p_{to} is the result of recursively translating the constructor with all but the first field. Note that the rules (ep-constr-3) and (ep-constr-4) are the only two rules in which variables are introduced into the patterns, and in both cases they are introduced into both resulting patterns, thereby maintaining the invariant that corresponding patterns contain the same variables.

The rule (ep-constr-5) is for multiple (i.e., at least two) constructors. In this case, we translate the first constructor C_1 on its own, resulting in a pair of patterns $p_{\text{from }1}; p_{\text{to }1}$. The other constructors are also recursively translated. In the representation type, the choice between C_1 and one of the other constructors is expressed by an application of Sum, where C_1 values correspond to *Inl* values, whereas values of any other constructor correspond to *Inr* values. Therefore, the pattern $p_{\text{to }1}$ is prefixed by *Inl*, and the remaining patterns $p_{\text{to }j}$ are all prefixed by *Inr*.

Theorem 10.2. Assume that K and Γ contain the bindings for Zero, Unit, Sum, and Prod, and their constructors. If D is a well-formed datatype, thus $\mathrm{K} \vdash D \rightsquigarrow T :: \kappa; \Gamma'$ where $\kappa \equiv \{\kappa_i \rightarrow\}^{i \in 1..n} *$, and if $[\![D]\!]^{\text{ep}} \equiv \mathsf{ep}(T) = e$ and $\mathsf{Str}(T) \equiv [\![D]\!]^{\text{str}}$, then

$$\mathrm{K}, T :: \kappa; \Gamma, \Gamma' \vdash e :: \{\forall a_i :: \kappa_i\}^{i \in 1..n}.\ \mathrm{EP}\ \big(T\ \{a_i\}^{i \in 1..n}\big)\ \big(\mathsf{Str}(T)\ \{a_i\}^{i \in 1..n}\big)\ .$$

The theorem states that the embedding-projection pair resulting from the declaration of a datatype is type correct and is indeed a value of type EP, being able to convert back and forth between values of type T and values of type $\mathsf{Str}(T)$, the structural representation of T.

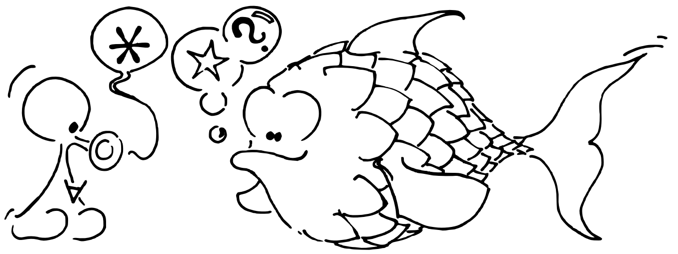
Theorem 10.3. Let D be a well-formed datatype, thus $\mathrm{K} \vdash D \rightsquigarrow T :: \kappa; \Gamma'$. Then, for any two expressions e and e', the FCR expression

```
let from = λz → case z of EP x y → x
    to   = λz → case z of EP x y → y
    ⟦D⟧^ep
in  (from ep(T) (to ep(T) e), to ep(T) (from ep(T) e'))
```

evaluates to (e, e').

Note that we include the *from* and *to* definitions in the let statement, because they may occur in the embedding-projection pair, but the definition of the embedding-projection pair, $[\![D]\!]^{\mathrm{ep}}$, is the important declaration.

11 Translation by Specialization



In Chapter 10, we have shown how a datatype can be converted into an isomorphic parametrized type that reveals the structure of the datatype definition. A generic function can be specialized to such a parametrized type expression using the algorithm discussed in Chapter 6.

In this Chapter we show how exactly the isomorphism between a datatype T and its generic representation type $\mathsf{Str}(T)$ can be exploited to reduce the specialization problem for a generic application $x\ \langle T\ \{A_i\}^{i\in 1..n}\rangle$ to the specialization problem for $x\ \langle \mathsf{Str}(T)\ \{A_i\}^{i\in 1..n}\rangle$. The solution has already been sketched in Section 7.5.2. In Section 11.1, we will discuss another example and show that the key to the problem is to lift the isomorphism between T and $\mathsf{Str}(T)$ that is available in an embedding-projection pair to an isomorphism between the types $t[T\ /\ a]$ and $t[\mathsf{Str}(T)\ /\ a]$ for a type t that is the base type of the generic function x. In Sections 11.2 and 11.3, we will show how this problem can be solved using a generic function called *bimap*. Afterwards, in Section 11.4, we will discuss implications for the dependency relation and show in Section 11.5 how generic functions can be translated into FCR. Section 11.6 adds some remarks on how to implement the translation. Having outlined how translation by specialization works, we discuss the merits of the approach and point out limitations in Section 11.7. We conclude by presenting two alternative possibilities to translate type-indexed and generic functions, in Section 11.8.

11.1 Problem

Let us recall the examples from Section 7.5.2. We were trying to specialize the calls *encode* $\langle \text{Tree}\ \alpha \rangle$ and *add* $\langle \text{Tree}\ \alpha \rangle$. The datatype Tree has been defined as

$$\textbf{data}\ \text{Tree}\ (a :: *) = \mathit{Leaf} \mid \mathit{Node}\ (\text{Tree}\ a)\ a\ (\text{Tree}\ a)\ ,$$

which makes the parametrized type

$$\Lambda a :: *.\ \text{Sum Unit}\ \Big(\text{Prod}\ (\text{Tree}\ a)\ \big(\text{Prod}\ a\ (\text{Tree}\ a)\big)\Big)$$

its structural representation. We will abbreviate as follows:

$$\mathsf{Str}(\text{Tree})\ A \equiv \text{Sum Unit}\ \Big(\text{Prod}\ (\text{Tree}\ A)\ \big(\text{Prod}\ A\ (\text{Tree}\ A)\big)\Big)\ .$$

The base types of the functions *add* and *encode* are

$$\begin{aligned}
&\mathsf{mbase}(\mathit{encode}\ \langle A \rangle) \equiv A \to [\text{Bit}]\\
&\mathsf{mbase}(\mathit{add}\ \ \ \ \ \langle A \rangle) \equiv A \to A \to A\ .
\end{aligned}$$

Now, we can specialize the calls *encode* $\langle \mathsf{Str}(\text{Tree})\ \alpha \rangle$ and *add* $\langle \mathsf{Str}(\text{Tree})\ \alpha \rangle$ – using the $[\![\cdot]\!]^{\text{gtrans}}$ algorithm from Figure 6.18 – to do the main part of the work, but we still need a wrapper for *encode* $\langle \text{Tree}\ \alpha \rangle$ and *add* $\langle \text{Tree}\ \alpha \rangle$, because Tree is recursive, and because the original call in the program refers to the Tree datatype, not to Str(Tree).

According to the gapp algorithm from Figure 9.12, we know that

$$\begin{aligned}
&\mathit{encode}\ \langle \mathsf{Str}(\text{Tree})\ \alpha \rangle :: \forall a.\ (\mathit{encode}\ \langle \alpha \rangle :: a \to [\text{Bit}]) \Rightarrow \mathsf{Str}(\text{Tree})\ a \to [\text{Bit}]\\
&\mathit{encode}\ \langle \text{Tree}\ \alpha \rangle\ \ \ \ \ \ \ \ \ :: \forall a.\ (\mathit{encode}\ \langle \alpha \rangle :: a \to [\text{Bit}]) \Rightarrow \text{Tree}\ a \ \ \ \ \ \ \ \to [\text{Bit}]\\
&\mathit{add}\ \ \ \ \ \langle \mathsf{Str}(\text{Tree})\ \alpha \rangle :: \forall a.\ (\mathit{add}\ \langle \alpha \rangle :: a \to a \to a)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\Rightarrow \mathsf{Str}(\text{Tree})\ a \to \mathsf{Str}(\text{Tree})\ a \to \mathsf{Str}(\text{Tree})\ a\\
&\mathit{add}\ \ \ \ \ \langle \text{Tree}\ \alpha \rangle\ \ \ \ \ \ \ \ \ :: \forall a.\ (\mathit{add}\ \langle \alpha \rangle :: a \to a \to a)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\Rightarrow \text{Tree}\ a \ \ \ \ \ \ \ \to \text{Tree}\ a \ \ \ \ \ \ \ \to \text{Tree}\ a\ .
\end{aligned}$$

We have an isomorphism between Str(Tree) t and Tree t for any type t. Therefore, also the types of the two *encode* applications as well as the types of the two *add* applications can be expected to be isomorphic. We have demonstrated by example that we can define the calls *encode* $\langle \text{Tree}\ \alpha\rangle$ and *add* $\langle \text{Tree}\ \alpha\rangle$ with the desired types by making use of the embedding-projection pair $\mathsf{ep}(\text{Tree})$ as follows:

$$\begin{aligned}
&\mathit{encode}\ \langle \text{Tree}\ \alpha\rangle\ x \quad = \mathit{encode}\ \langle \mathsf{Str}(\text{Tree})\ \alpha\rangle\ (\mathit{from}\ \mathsf{ep}(\text{Tree})\ x)\\
&\mathit{add}\quad\ \langle \text{Tree}\ \alpha\rangle\ x\ y =\\
&\quad \mathit{to}\ \mathsf{ep}(\text{Tree})\ \big(\mathit{add}\ \langle \mathsf{Str}(\text{Tree})\ \alpha\rangle\ (\mathit{from}\ \mathsf{ep}(\text{Tree})\ x)\ (\mathit{from}\ \mathsf{ep}(\text{Tree})\ y)\big)\ .
\end{aligned}$$

The question remains – can we find a general solution to the following problem: we have a named type $T\ \{\alpha_i\}^{i\in 1..n}$, a generic representation type thereof, abbreviated as $\mathsf{Str}(T)\ \{\alpha_i\}^{i\in 1..n}$, and a generic function x. We must find a way to define $x\ \langle T\ \{\alpha_i\}^{i\in 1..n}\rangle$ in terms of $x\ \langle \mathsf{Str}(T)\ \{\alpha_i\}^{i\in 1..n}\rangle$, making use of $\mathsf{ep}(T)$.

11.2 Lifting isomorphisms

As it turns out, we can solve the problem just posed. Assume for now that x is a type-indexed function of arity $\langle r \mid 0\rangle$ for some r, i.e., it has no non-generic variables. We can then define

$$\begin{aligned}
&x\ \langle T\ \{\alpha_j\}^{j\in 1..n}\rangle =\\
&\quad \mathbf{let}\ \{\mathit{bimap}\ \langle \beta_i\rangle = \mathsf{ep}(T)\}_{;}^{i\in 1..r}\\
&\quad \mathbf{in}\ \ \mathit{to}\ \mathit{bimap}\ \big\langle \mathsf{mbase}(x\ \langle\{\beta_i\}_{,}^{i\in 1..r}\rangle)\big\rangle\ \big(x\ \langle \mathsf{Str}(T)\ \{\alpha_j\}^{j\in 1..n}\rangle\big)\ .
\end{aligned}$$

In above definition, we use local redefinition on a generic function *bimap* yet to be defined. This function *bimap* is a bidirectional variant of *map* (defined on page 135 in Section 9.1), and computes embedding-projection pairs instead of functions. While introducing *map*, we noticed a strong relationship between the identity function and mapping, and that we can use local redefinition to “plug in” a modifying function into an identity that works on a complicated datatype. Here, we use the same idea, only that the function that we plug in is the embedding-projection pair $\mathsf{ep}(T)$. The type we use *bimap* on is the base type of x, instantiated to dependency variables β_i in the generic slots, because we want to change all occurrences of the type arguments from an application of $\mathsf{Str}(T)$ to an application of T.

The definition of *bimap* is indeed strongly related to *map*. The type signature is

$$\mathit{bimap}\ \langle a_1 :: *, a_2 :: *\rangle :: (\mathit{bimap}\ \langle a_1, a_2\rangle) \Rightarrow \text{EP}\ a_1\ a_2\ .$$

The function *bimap* results in an EP where *map* uses the function space constructor $(\to)$. Instead of defining one function, from a type a_1 to a type a_2, *bimap* defines a pair of functions, one from a_1 to a_2, the other from a_2 to a_1. The advantage is that it is straightforward to extend the definition of *bimap* to function types. This is the definition of the generic function:

```
bimap ⟨Unit⟩      = EP id id
bimap ⟨Sum α β⟩ =
  let fromSum z = case z of
                    Inl x → Inl (from (bimap ⟨α⟩) x)
                    Inr x → Inr (from (bimap ⟨β⟩) x)
      toSum z   = case z of
                    Inl x → Inl (to   (bimap ⟨α⟩) x)
                    Inr x → Inr (to   (bimap ⟨β⟩) x)
  in  EP fromSum toSum
bimap ⟨Prod α β⟩ =
  let fromProd z = case z of
                    x × y → (from (bimap ⟨α⟩) x) × (from (bimap ⟨β⟩) y)
      toProd z   = case z of
                    x × y → (to   (bimap ⟨α⟩) x) × (to   (bimap ⟨β⟩) y)
  in  EP fromProd toProd
bimap ⟨α → β⟩    =
  let fromFun z  = from (bimap ⟨β⟩) · z · to   (bimap ⟨α⟩)
      toFun z    = to   (bimap ⟨β⟩) · z · from (bimap ⟨α⟩)
  in  EP fromFun toFun
```

In addition to the arms given above, *bimap* can be defined as *EP id id* for all abstract types of kind $*$, including Int, Float and Char.

We also need to define *bimap* on EP, to close a vicious circle: if we do not, the automatic specialization would produce the following definition:

```
bimap ⟨EP α β⟩ = let bimap ⟨γ⟩ = ep(EP)
                 in  to bimap ⟨EP γ γ⟩ (bimap ⟨Prod (α → β) (β → α)⟩) .
```

This definition calls itself immediately (the let does not matter, and *to* is strict in its argument), thus a call of *bimap* to EP would not terminate. Here is a definition that works:

```
bimap ⟨EP α β⟩ =
  let e₁       = bimap ⟨α → β⟩
      e₂       = bimap ⟨β → α⟩
      fromEP z = case z of
                  EP x y → EP (from e₁ x) (from e₂ y)
```

```
        toEP z    = case z of
                         EP x y → EP (to   e₁ x) (to   e₂ y)
      in EP fromEP toEP .
```

The following theorem states that if we proceed as outlined above, specializing a call to a generic function with a new datatype by producing a wrapper around the call that uses the structural representation of that datatype, we always produce a type correct component.

Theorem 11.1. For a type-indexed function x of arity $\langle r \mid 0\rangle$, the declaration

$$\begin{aligned}
&x\ \langle T\ \{\alpha_j\}^{j\in 1..n}\rangle = \\
&\quad \textbf{let}\ \{\mathit{bimap}\ \langle\beta_i\rangle = \mathsf{ep}(T)\}_{;}^{i\in 1..r} \\
&\quad \textbf{in}\ \ \mathit{to}\ \mathit{bimap}\ \big\langle \mathsf{mbase}(x\ \langle\{\beta_i\}_{,}^{i\in 1..r}\rangle)\big\rangle\ \big(x\ \langle \mathsf{Str}(T)\ \{\alpha_j\}^{j\in 1..n}\rangle\big)
\end{aligned}$$

is type correct, i.e., the right hand side of the declaration is of the qualified type $\mathsf{gmapp}(x\ \langle T\ \{\alpha_j\}^{j\in 1..n}\rangle)$.

Proof. According to the revised generic application algorithm from Figure 9.12, we get that

$$\begin{aligned}
&\mathsf{gmapp}(x\ \langle T\ \{\alpha_i\}^{j\in 1..n}\rangle) \equiv \\
&\quad \big\{\{\forall a_{i,j} :: \kappa_j.\}^{j\in 1..n}\big\}^{i\in 1..r}\ (\Delta) \Rightarrow \mathsf{mbase}\big(x\ \langle\{T\ \{a_{i,j}\}^{j\in 1..n}\}_{,}^{i\in 1..r}\rangle\big)
\end{aligned}$$

for a set of dependency constraints Δ containing dependencies for all combinations of functions in $\mathsf{dependencies}(x)$ and variables α_i. The above qualified type is the type that the right hand side expression should have.

The generic application algorithm also dictates that

$$\begin{aligned}
&x\ \langle \mathsf{Str}(T)\ \{\alpha_i\}^{j\in 1..n}\rangle :: \\
&\quad \big\{\{\forall a_{i,j} :: \kappa_j.\}^{j\in 1..n}\big\}^{i\in 1..r}\ (\Delta) \Rightarrow \mathsf{mbase}\big(x\ \langle\{\mathsf{Str}(T)\ \{a_{i,j}\}^{j\in 1..n}\}_{,}^{i\in 1..r}\rangle\big)\ ,
\end{aligned}$$

for the same Δ. The only difference between this type and the type above is that one uses $\mathsf{Str}(T)$ where the other uses T.

The local redefinition extends Δ to Δ' for the scope of the let. We have

$$\vdash \mathsf{ep}(T) :: \{\forall a_i :: \kappa_i.\}^{i\in 1..n}\ \mathrm{EP}\ (T\ \{a_i\}^{i\in 1..n})\ (\mathsf{Str}(T)\ \{a_i\}^{i\in 1..n})\ ,$$

thus Δ' is Δ extended with

$$\big\{\mathit{bimap}\ \langle\beta_i\rangle :: \{\forall a_{i,j} :: \kappa_j.\}^{j\in 1..n}\ \mathrm{EP}\ (T\ \{a_{i,j}\}^{j\in 1..n})\ (\mathsf{Str}(T)\ \{a_{i,j}\}^{j\in 1..n})\big\}^{i\in 1..r}\ .$$

Another application of gmapp, for the *bimap* call, yields

$$\begin{aligned}
&\vdash \mathit{bimap}\ \langle \mathsf{mbase}(x\ \langle\{\beta\}_{,}^{i\in 1..r}\rangle)\rangle\\
&\quad :: \{\forall(b_{i,1} :: *)\ (b_{i,2} :: *)\}^{i\in 1..r}.)\ \big(\{\mathit{bimap}\ \langle\beta_i :: *\rangle :: \mathrm{EP}\ b_{i,1}\ b_{i,2}\}_{,}^{i\in 1..r}\big)\\
&\qquad \Rightarrow \mathrm{EP}\ \big(\mathsf{mbase}(x\ \langle\{b_{i,1}\}_{,}^{i\in 1..r}\rangle)\big)\ \big(\mathsf{mbase}(x\ \langle\{b_{i,2}\}_{,}^{i\in 1..r}\rangle)\big)\ .
\end{aligned}$$

Considering Δ', we can conclude that

$$\begin{aligned}
&\Delta' \vdash \mathit{to\ bimap}\ \langle \mathsf{mbase}(x\ \langle\{\beta\}_{,}^{i\in 1..r}\rangle)\rangle\\
&\quad :: \mathsf{mbase}\big(x\ \langle\{\mathsf{Str}(T)\ \{a_{i,j}\}^{j\in 1..n}\}_{,}^{i\in 1..r}\rangle\big) \to \mathsf{mbase}\big(x\ \langle\{T\ \{a_{i,j}\}^{j\in 1..n}\}_{,}^{i\in 1..r}\rangle\big)
\end{aligned}$$

and

$$\Delta' \vdash x\ \langle \mathsf{Str}(T)\ \{\alpha_i\}^{j\in 1..n}\rangle :: \mathsf{mbase}\big(x\ \langle\{\mathsf{Str}(T)\ \{a_{i,j}\}^{j\in 1..n}\}_{,}^{i\in 1..r}\rangle\big)$$

for suitable $a_{i,j}$. After applying (e-app) to the two terms and revealing the dependency constraints, we are done. □

11.3 Lifting isomorphisms and universal quantification

In the previous section, we assumed that the arity of the generic function that in the call is $\langle r \mid 0\rangle$ for some r. Now, we turn to the general case where the arity is $\langle r \mid s\rangle$ for natural numbers r and s.

The non-generic slots of a base type are usually universally quantified on the outside; therefore one possible way to generate a wrapper would be to define

$$\begin{aligned}
&x\ \langle T\ \{\alpha_j\}^{j\in 1..n}\rangle =\\
&\quad \mathbf{let}\ \{\mathit{bimap}\ \langle\beta_i\rangle = \mathsf{ep}(T)\}_{;}^{i\in 1..r}\\
&\quad \mathbf{in}\ \ \mathit{to\ bimap}\ \Big\langle \{\forall b_j :: \kappa_j.\}^{j\in 1..s}\ \mathsf{mbase}\big(x\ \langle\{\beta_i\}_{,}^{i\in 1..r} \mid \{b_j\}_{,}^{j\in 1..s}\rangle\big)\Big\rangle\\
&\qquad\qquad \big(x\ \langle \mathsf{Str}(T)\ \{\alpha_j\}^{j\in 1..n}\rangle\big)
\end{aligned}$$

where the b_j are fresh type variables of suitable kind. However, universal quantifications are not allowed in the language of type arguments, and it is not clear how we should specialize the *bimap* call in the code fragment above.

We will use this as incentive to discuss universal quantification over kind $*$ type variables and outline two possible solutions. Subsequently, we discuss why these approaches are difficult to extend to variables of higher kind.

After that, we use the knowledge gained to circumvent the direct need for universal quantification in type arguments in the situation of *bimap* above.

11.3.1 Kind $*$ default arms

A possible extension to our language is to allow a default arm of a generic function for types of kind $*$ that do not have an explicit arm. Such an extension can be useful for several of the functions that we have encountered so far:

$$\begin{aligned}&\mathit{size}\ \langle\_ :: *\rangle\ x = 0\\ &\mathit{map}\ \langle\_ :: *\rangle\ x = x\\ &\mathit{collect}\ \langle\_ :: *\rangle\ x = [\,]\end{aligned}$$

In the case of the generic functions *map*, *zipWith* and *collect*, this makes the explicit definition of Unit and all kind $*$ abstract types such as Int, Float, and Char superfluous. It not only saves code lines – the function also becomes applicable to any other abstract type that might be introduced into the program.

The kind $*$ default must be sufficiently polymorphic – it must not make any assumptions about the type. This can be achieved by extending gapp to handle kind $*$ arms as follows:

$$\mathsf{gapp}(x\ \langle\_ :: *\rangle) \equiv \forall a :: *.\ \mathsf{gapp}(x\ \langle a\rangle)\ \ .$$

Subsequently, whenever we have to specialize a call $x\ \langle T\rangle$ for a named type T of kind $*$, we first check if T is in the signature of T. If so, we use the explicit arm. Otherwise, we check if there is a kind $*$ default, and use that if possible. Only if no kind $*$ default and no explicit arm exist, we try to generate a component using the generic specialization technique outlined in Sections 11.1 and 11.2.

Kind $*$ default arms also help with universally quantified type variables of kind $*$, because they work for *all* types of kind $*$, even for a type we do not know. When encountering a call $x\ \langle\forall a :: *.\ A\rangle$, we can thus skolemize a to a fresh named type T, and treat the call as $x\ \langle A[T\ /\ a]\rangle$. As long as there is a kind $*$ default for x, this will be used during the specialization.

For *bimap*, we can define a kind $*$ default arm as follows:

$$\mathit{bimap}\ \langle a :: *\rangle = \mathit{EP}\ \mathit{id}\ \mathit{id}\ \ .$$

This will work as intended and makes the explicit definitions for Unit and the abstract types unnecessary.

11.3.2 Arms for universal quantification

A quantification over a kind $*$ variable can be viewed as application of a type constructor of kind $(* \to *) \to *$. If we have a value of type

$$\forall a :: *.\ t$$

and assume that we can somehow rewrite it to a type where the occurrences of a in t are explicit, such as

$$\forall a :: *.\ t'\ a$$

with $a \notin \mathsf{ftv}(t')$, then the application of the quantifier is a type-level function that takes the type t', of kind $* \to *$, to the type $\forall a :: *.\ t'\ a$, of kind $*$. In Haskell with extensions, this idea could be captured by defining a type synonym

$$\textbf{type}\ \text{Forall}_*\ (c :: * \to *) = \forall a :: *.\ c\ a\ \ .$$

Now, if universal quantification over a kind $*$ datatype is just the application of a type constructor, then type-indexed functions can be defined for this type.

$$\begin{aligned}
&\mathit{size}\ \ \ \langle \forall a :: *.\ \gamma\ a\rangle\ x = \textbf{let}\ \mathit{size}\ \langle\alpha\rangle = 0\\
&\qquad\qquad\qquad\qquad\quad\ \textbf{in}\ \ \mathit{size}\ \langle\gamma\ \alpha\rangle\ x\\
&\mathit{map}\ \ \ \langle \forall a :: *.\ \gamma\ a\rangle\ x = \textbf{let}\ \mathit{map}\ \langle\alpha\rangle = \mathit{id}\\
&\qquad\qquad\qquad\qquad\quad\ \textbf{in}\ \ \mathit{map}\ \langle\gamma\ \alpha\rangle\ x\\
&\mathit{collect}\ \langle \forall a :: *.\ \gamma\ a\rangle\ x = \textbf{let}\ \mathit{collect}\ \langle\alpha\rangle = [\,]\\
&\qquad\qquad\qquad\qquad\quad\ \textbf{in}\ \ \mathit{collect}\ \langle\gamma\ \alpha\rangle\ x\ \ .
\end{aligned}$$

In all cases, we essentially know how to define the function for any kind $*$ datatype, thus also for a universally quantified one.

Other than in the solution using a kind $*$ default, we have more control over the structure of the type. We can, for example, write a generic function that counts the universal quantifiers in the type:

$$\begin{aligned}
&\mathit{countForall}\ \langle a :: *\rangle && :: (\mathit{countForall}\ \langle a\rangle) \Rightarrow \text{Int}\\
&\mathit{countForall}\ \langle \text{Int}\rangle && = 0\\
&\mathit{countForall}\ \langle \text{Unit}\rangle && = 0\\
&\mathit{countForall}\ \langle \text{Sum}\ \alpha\ \beta\rangle && = \mathit{countForall}\ \langle\alpha\rangle + \mathit{countForall}\ \langle\beta\rangle\\
&\mathit{countForall}\ \langle \text{Prod}\ \alpha\ \beta\rangle && = \mathit{countForall}\ \langle\alpha\rangle + \mathit{countForall}\ \langle\beta\rangle\\
&\mathit{countForall}\ \langle \alpha \to \beta\rangle && = \mathit{countForall}\ \langle\alpha\rangle + \mathit{countForall}\ \langle\beta\rangle\ ,\\
&\mathit{countForall}\ \langle \forall a :: *.\ \gamma\ a\rangle && = 1 + \textbf{let}\ \mathit{countForall}\ \langle\alpha\rangle = 0\\
& && \qquad\ \ \textbf{in}\ \ \mathit{countForall}\ \langle\gamma\ \alpha\rangle
\end{aligned}$$

and then call

$$\mathit{countForall}\ \langle(\forall a :: *.\ a \to \text{Int} \to a) \to (\forall a :: *.\ \text{Int} \to a \to a)\rangle$$

to get the result 2. Similarly, we could write a *show* function that displays universal quantification in a human-readable fashion, assigning a fresh variable name from a supply of names whenever a universal quantifier is encountered. These effects are out of reach using kind $*$ default definitions.

Of course, the definition of *countForall* demonstrates that it might be useful to have both kind $*$ defaults and arms for universal quantification. We could augment the definition of *countForall* using

$$\mathit{countForall}\ \langle \_ :: * \rangle = 0\ ,$$

discarding the arms for Int and Unit instead. Having both is not a problem – the two extensions can be added to the language independently.

For *bimap*, we can use the definition:

$$\begin{aligned}\mathit{bimap}\ \langle \forall a :: *.\ \gamma\ a \rangle = \mathbf{let}\ & \mathit{bimap}\ \langle \alpha \rangle = \mathit{EP}\ \mathit{id}\ \mathit{id} \\ \mathbf{in}\ & \mathit{bimap}\ \langle \gamma\ \alpha \rangle\ .\end{aligned}$$

Note that the type of this arm is

$$\begin{aligned}&\forall (c_1 :: * \to *)\ (c_2 :: * \to *). \\ &\quad \big(\mathit{bimap}\ \langle \gamma\ \alpha \rangle :: \forall (a_1 :: *)\ (a_2 :: *).\ (\mathit{bimap}\ \langle \alpha \rangle :: \mathrm{EP}\ a_1\ a_2) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \mathrm{EP}\ (c_1\ a_1)\ (c_2\ a_2)\big) \\ &\qquad \Rightarrow \mathrm{EP}\ (\forall a :: *.\ c_1\ a)\ (\forall a :: *.\ c_2\ a)\ .\end{aligned}$$

This type is valid in our functional core language, but it is not valid in Haskell – not even with the current set of extensions available, because it uses the type constructor EP impredicatively at a polymorphic type. There is a GHC extension for arbitrary rank types (Peyton Jones and Shields 2003), but it allows polymorphic types only as arguments to the function space constructor $(\to)$. Other type constructors, such as EP, may only be applied to monomorphic types.

Therefore, this solution works only if we either make it specific to the generic function *bimap* and take special precautions for the polymorphic case (for example, use another, monomorphic type, which is allowed to contain polymorphic fields in GHC), or if we really have support for type constructors to be instantiated with polymorphic values in our target language – at least in some limited form. For instance, if we would allow tuples to contain polymorphic values, we could define EP as a type synonym

$$\mathbf{type}\ \mathrm{EP}\ (a :: *)\ (b :: *) = (a \to b, b \to a)$$

which would be sufficient for our purposes.

Yet another possibility is to define *bimap* in two parts, *from* and *to*, where *from* and *to* are both defined in exactly the same way as *map*, but have an additional case for function types that makes use of the other function:

$$\begin{array}{lll} \mathit{from}\ \langle a :: *, b :: * \rangle & :: & (\mathit{from}\ \langle a,b \rangle, \mathit{to}\ \langle b,a \rangle) \Rightarrow a \to b \\ \mathit{to}\ \ \ \ \langle b :: *, a :: * \rangle & :: & (\mathit{from}\ \langle a,b \rangle, \mathit{to}\ \langle b,a \rangle) \Rightarrow b \to a \end{array}$$

$$\begin{array}{ll} \mathit{from}\ \langle \alpha \to \beta \rangle\ z & = \mathit{from}\ \langle \beta \rangle \cdot z \cdot \mathit{to}\ \ \ \ \langle \alpha \rangle \\ \mathit{to}\ \ \ \ \langle \alpha \to \beta \rangle\ z & = \mathit{to}\ \ \ \ \langle \beta \rangle \cdot z \cdot \mathit{from}\ \langle \alpha \rangle \end{array}$$

If we now extend *from* and *to* with cases for universal quantifiers, we still get a higher ranked type, but the quantifiers appear in positions where they are legal in GHC.

Even if the problem of assigning a type to the universal case is solved, there is still the question of how to express a call of the form $x\ \langle \forall a :: *.\ A \rangle$ for some type argument A, possibly containing a, in terms of the case $x\ \langle \forall a :: *.\ \gamma\ a \rangle$ which requires a type constructor of kind $* \to *$ that expects the quantified variable as an argument. This abstraction step can be simulated using local redefinition: we can define

$$\begin{array}{ll} x\ \langle \forall a :: *.\ A \rangle \equiv & \mathbf{let}\ \{y_i\ \langle \gamma\ (\alpha :: *) \rangle = y_i\ \langle A[\alpha\ /\ a] \rangle\}_{;}^{i \in 1..k} \\ & \mathbf{in}\ \ x\ \langle \forall a :: *.\ \gamma\ a \rangle\ , \end{array}$$

where $\{y_i\}_{,}^{i \in 1..k}$ are the dependencies of x. This translation is only type correct if either the y_i are all reflexive, i.e., depend on themselves, or if we exclude the case that $A \equiv a$. The reflexivity requirement originates from the fact that γ would essentially be instantiated by the identity type constructor $\Lambda a :: *.\ a$ if $A \equiv a$. Reflexivity will be treated in more detail in the Section 11.4. For *bimap*, there is no problem, because *bimap* depends only on itself, and is thus reflexive.

11.3.3 Quantified variables of higher kind

Unfortunately, the two solutions just discussed do not help with universally quantified variables of higher kind. The problem is that in Haskell, we cannot write a *map* function that is polymorphic in the type constructor:

$$\mathit{pmap} :: \forall (a :: *)\ (b :: *)\ (f :: * \to *).\ (a \to b) \to (f\ a \to f\ b)$$

But such a function is exactly what would be needed to define a case such as

$$\mathit{map}\ \langle \forall b :: * \to *.\ \gamma\ b \rangle\ x = \ \ldots$$

or even

$$\mathit{bimap}\ \langle \forall b :: * \to *.\ \gamma\ b \rangle\ x = \ \ldots\ \ .$$

The reason why we cannot define *pmap* is that we know nothing about the structure of f, and how to map a function over a value of type $f\ a$ for some type a

depends on the structure of the type constructor f. This is the reason why we defined a generic function *map* in the first place.

There are possibilities to work around this problem, for instance to allow dependency constraints *within* type patterns:

$$\begin{array}{l} map\ \langle \forall b :: * \to *. \\ \quad (map\ \langle b\ \alpha \rangle :: \forall a_1\ a_2.(map\ \langle \alpha \rangle \Rightarrow (a_1 \to a_2) \to b\ a_1 \to b\ a_2)) \\ \qquad \Rightarrow \gamma\ b \rangle\ x = \mathbf{let}\ map\ \langle \beta\ (\alpha :: *) \rangle = map\ \langle b\ \alpha \rangle \\ \qquad\qquad\qquad\quad \mathbf{in}\ \ map\ \langle \gamma\ \beta \rangle\ . \end{array}$$

Note that we also need the name b in scope on the right hand side. Here, we expect that with the quantified type constructor b comes some information of how to map over it. We can then put this information to use. We could define a *bimap* case similarly.

But if we can somehow give semantics to this case, it still does not work for all universal quantifications, but only those that provide a *map* dependency of the required form. For our *bimap* situation, that would mean that non-generic variables of higher kind must not be simply universally quantified, but also be equipped with mapping information.

Furthermore, all this only covers kind $* \to *$ so far. For other kinds, we would need to introduce yet more cases, and there is no easy way to generalize to universally quantified type variables of arbitrary kind.

11.3.4 A pragmatic solution for *bimap*

If we do not want to extend the language with either kind $*$ default cases or cases for universal quantification, and at the same time want to prevent problems with impredicativity, we can use local redefinition once more in the definition of the wrapper function.

We define

$$\begin{array}{l} x\ \langle T\ \{\alpha_j\}^{j \in 1..n} \rangle = \\ \quad \mathbf{let}\ \{bimap\ \langle \beta_i \rangle = \mathsf{ep}(T)\}_{;}^{i \in 1..r} \\ \qquad\quad \{bimap\ \langle \beta'_j \rangle = EP\ id\ id\}_{;}^{j \in 1..s} \\ \quad \mathbf{in}\ \ to\ bimap\ \big\langle \mathsf{mbase}(x\ \langle \{\beta_i\}_{,}^{i \in 1..r} \mid \{\beta'_j\}_{,}^{j \in 1..s} \rangle) \big\rangle \\ \qquad\qquad \big(x\ \langle \mathsf{Str}(T)\ \{\alpha_j\}^{j \in 1..n} \rangle\big) \end{array}$$

and directly plug in *EP id id* for all occurrences of non-generic type variables – assuming they are all of kind $*$ – thus alleviating the need for universal quantification.

The same, namely using *EP id id* in a local redefinition, works also for non-generic type variables of higher kind, as long as they are not applied to the type argument, but to a constant type. Then, the wrapper need not map the conversion between $\mathsf{Str}(T)$ and T over an unknown type constructor, and there is no problem. We disallow situations where non-generic variables are applied to the type argument.

Theorem 11.2. For a type-indexed function x of arity $\langle r \mid s\rangle$, where all non-generic slots of x are of kind $*$, the declaration

$$\begin{array}{l} x\ \langle T\ \{\alpha_j\}^{j\in 1..n}\rangle = \\ \quad \mathbf{let}\ \{\mathit{bimap}\ \langle\beta_i\rangle = \mathsf{ep}(T)\}_{;}^{i\in 1..r} \\ \qquad\ \ \{\mathit{bimap}\ \langle\beta_j'\rangle = \mathit{EP\ id\ id}\}_{;}^{j\in 1..s} \\ \quad \mathbf{in}\ \ \mathit{to\ bimap}\ \big\langle \mathsf{mbase}(x\ \langle\{\beta_i\}_{,}^{i\in 1..r} \mid \{\beta_j'\}_{,}^{j\in 1..s}\rangle)\big\rangle \\ \qquad\qquad \big(x\ \langle\mathsf{Str}(T)\ \{\alpha_j\}^{j\in 1..n}\rangle\big) \end{array}$$

is type correct, i.e., the right hand side of the declaration is of the qualified type $\mathsf{gmapp}(x\ \langle T\ \{\alpha_j\}^{j\in 1..n}\rangle)$.

The proof is largely identical to the proof of Theorem 11.1.

11.4 Reflexivity of the dependency relation

It turns out that we also need an additional requirement on the dependencies for generic functions. Consider the following datatype:

$$\mathbf{data}\ \mathrm{Id}\ (a :: *) = \mathit{Id}\ a\ \ .$$

According to the translations defined in Chapter 10, we have

$$[\![\mathbf{data}\ \mathrm{Id} = \Lambda a :: *.\ \mathit{Id}\ a]\!]^{\mathrm{str}} \equiv \Lambda a :: *.\ a$$

and

$$\begin{array}{llll} [\![\mathbf{data}\ \mathrm{Id} = \Lambda a :: *.\ \mathit{Id}\ a]\!]^{\mathrm{ep}} \equiv \mathsf{ep}(\mathrm{Id}) = & \mathbf{let}\ x_{\mathrm{from}} & = \lambda y \to \mathbf{case}\ y\ \mathbf{of} & \\ & & \qquad\qquad \mathrm{Id}\ x \to x & \\ & \qquad\ x_{\mathrm{to}} & = \lambda y \to \mathbf{case}\ y\ \mathbf{of} & \\ & & \qquad\qquad x \quad\ \to \mathrm{Id}\ x & \\ & \mathbf{in}\ \ \mathit{EP}\ x_{\mathrm{from}}\ x_{\mathrm{to}}\ \ . & & \end{array}$$

We will write $\mathsf{Str}(\mathrm{Id})\ A$ for A.

Now, if a call such as $x \langle \mathrm{Id}\ \alpha \rangle$ is encountered in the program, for a generic function x that has no specific arm for Id, we have seen that we could make use of the definition

$$\begin{aligned} x\ \langle \mathrm{Id}\ \alpha \rangle = {} & \mathbf{let}\ \mathit{bimap}\ \langle \beta :: * \rangle = \mathsf{ep}(\mathrm{Id}) \\ & \mathbf{in}\ \ \mathit{to}\ \mathit{bimap}\ \langle \mathsf{base}(x\ \langle \beta \rangle) \rangle\ (x\ \langle \mathsf{Str}(\mathrm{Id})\ \alpha \rangle)\ , \end{aligned}$$

which, reducing the abbreviation $\mathsf{Str}(\mathrm{Id})\ \alpha$ to α, is

$$\begin{aligned} x\ \langle \mathrm{Id}\ \alpha \rangle = {} & \mathbf{let}\ \mathit{bimap}\ \langle \beta :: * \rangle = \mathsf{ep}(\mathrm{Id}) \\ & \mathbf{in}\ \ \mathit{to}\ \mathit{bimap}\ \langle \mathsf{base}(x\ \langle \beta \rangle) \rangle\ (x\ \langle \alpha \rangle)\ \ . \end{aligned}$$

Now, the right hand side refers to $x\ \langle \alpha \rangle$, and therefore depends on $x\ \langle \alpha \rangle$, but $x\ \langle \alpha \rangle$ is available only if x depends on itself!

We have thus seen that transforming a datatype to its structural representation as defined in Chapter 10, and subsequently applying the translation using the lifted isomorphism by locally redefining *bimap*, can lead to a translation which requires that a generic function depends on itself. We call a function that depends on itself **reflexive**.

Note that Id is not the only datatype that triggers this problem. Another example is the datatype for fixpoints on the type level:

$$\mathbf{data}\ \mathrm{Fix}\ (a :: * \to *) = \mathit{In}\ (a\ (\mathrm{Fix}\ a))\ \ .$$

Its structural representation is the parametrized type

$$\Lambda a :: * \to *.\ a\ (\mathrm{Fix}\ a)\ \ .$$

The situation that we need a reflexive dependency arises whenever one of the type variables appears in the head of the representation type. That – given the translation we use to compute the structural representation – happens whenever the datatype in question has exactly one constructor with exactly one field, and a type variable is the head of the type of this field.

There are basically two solutions to this problem: first, we can live with the problem and accept the additional requirement that a function can only be *generic* if it is reflexive. A non-reflexive type-indexed function can still exist, but it is not possible to specialize it to datatypes generically.

Second, we can modify the representation type generation in such a way that the problem does not arise. We could, for example, introduce Type as yet another special datatype and use it in the translation of all datatypes. The adapted rule (str-data) from Figure 10.2 would look as follows:

$$\frac{[\![\{(C_j\ \{t_{j,k}\}^{k \in 1..n_j})\}_{|}^{j \in 1..m}]\!]^{\mathrm{str}} \equiv t}{\begin{aligned} & [\![\mathbf{data}\ T = \{\Lambda a_i :: \kappa_i.\}^{i \in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}]\!]^{\mathrm{str}} \\ & \qquad\qquad \equiv \{\Lambda a_i :: \kappa_i.\}^{i \in 1..\ell}\ \mathrm{Type}\ t \end{aligned}}$$

Although using an explicit marker such as Type for the border of a datatype has certain advantages (and we will revisit this possibility in Chapter 17), it also has the disadvantage that every single generic function would need an additional case for Type, one that is usually trivial, but tedious to write.

As a variation of the second solution, we could adapt the structural type representations in such a way that we introduce the Id type into the representations in only those situations where one of the type variables appears in the head. We could then allow automatic specialization of a generic function to Id if the function is reflexive, and report a specialization error instead.

On the other hand, there are very few generic functions that are not reflexive anyway – with the exception of generic abstractions, which will be handled in Chapter 12. Therefore, we go for the first solution, which is the simplest: in the formal translation outlined in the following Section 11.5, a function can only be *generic*, i.e., new components can only be generated, if the function is reflexive.

11.5 Translation of generic functions

In this section, we will adapt the translation rules in such a way that reflexive functions that are defined for at least some of the datatypes Unit, Sum, Prod, and Zero, can be extended in a generic way to a large class of datatypes.

First of all, we have to make the information that is generated by the translations described in Chapter 10 available where it is needed: the embedding-projection pairs must be included as declarations into the translation of the program so that they can be referred to from everywhere; the information about the structural representations must be passed in an environment to all places where generic functions can be defined.

Furthermore, we have to assume that the generic *bimap* function is available, because we need it to specialize calls to generic functions.

Apart from that, we do not need to change a lot. At the declaration site of a generic function, we now make it possible to generate additional components for a generic function, and thereby to extend the signature environment in the scope of the generic function.

The language that we define now is called FCR+gf, and includes all the features we have seen so far, such as type-indexed functions with parametrized type patterns, dependencies, and local redefinition.

To distribute information, we introduce two new environments. The environment E for declarations of the embedding-projection pairs contains entries of the form $\mathsf{ep}(T) = d$. The structural representation types are stored in Ψ, using entries of the form $\mathsf{Str}(T) \equiv \{\Lambda a_i :: \kappa_i .\}^{i \in 1..\ell}\ t$. We assume that the number of entries in Ψ

and E is always the same. For a type T, there exists an embedding-projection pair if and only if there is a structural representation type. Utilizing this assumption, we need E only for checking programs, not in the judgments for declarations and expressions.

$$[\![P :: t]\!]^{\mathrm{gf}}_{\mathrm{K};\Gamma;\Psi;\mathrm{E}} \equiv P'$$

$$\frac{\begin{array}{c} P \equiv \{D_i;\}^{i \in 1..n}\ \mathbf{main} = e \\ \{\mathrm{K} \vdash D_i \rightsquigarrow \mathrm{K}_i; \Gamma_i\}^{i \in 1..n} \\ \mathrm{K} \equiv \mathrm{K}_0\ \{, \mathrm{K}_i\}^{i \in 1..n} \quad \Gamma \equiv \Gamma_0\ \{, \Gamma_i\}^{i \in 1..n} \\ \Psi \equiv \Psi_0\ \{, [\![D_i]\!]^{\mathrm{str}}\}^{i \in 1..n} \quad \mathrm{E} \equiv \mathrm{E}_0\ \{, [\![D_i]\!]^{\mathrm{ep}}\}^{i \in 1..n} \\ [\![\mathbf{let}\ d_{\mathrm{from}}; d_{\mathrm{to}}\ \mathbf{in}\ \mathbf{let}\ d_{\mathrm{bimap}}\ \mathbf{in}\ e :: t]\!]^{\mathrm{gf}}_{\mathrm{K};\Gamma;\varepsilon;\varepsilon;\Psi} \equiv e' \\ P' \equiv \{D_i;\}^{i \in 1..n}\ \mathbf{main} = \mathbf{let}\ \mathrm{E}\ \mathbf{in}\ e' \end{array}}{[\![P :: t]\!]^{\mathrm{gf}}_{\mathrm{K}_0;\Gamma_0;\Psi_0;\mathrm{E}_0} \equiv P'} \quad \text{(p-gprog)}$$

Figure 11.1: Well-formed programs, replaces Figure 3.7

Both environments, Ψ and E, are populated while checking and translating the program. We present a new rule to check the well-formedness of a program while at the same time translating it, in Figure 11.1. This judgment is a replacement for Figure 3.7, which was intended for language FCR. We have already extended that original judgment in a trivial way to be able to deal with dependencies in Chapter 6, without repeating the rule.

The new rule is named (p-gprog), to emphasize that it is capable of handling generic functions. We take four input environments, containing external types with their kinds, external functions and constructors (as in Chapter 3, if we import external functions, we have to extend the set of reduction rules for FCR as well), and now also structural representation types and embedding-projection pairs for external types.

We assume that Unit, Sum, Prod, and Zero are all contained in K_0, with their constructors in Γ_0. These types have neither structural representations nor embedding-projection pairs in Ψ_0 and E_0 – they are abstract.

On the other hand, the datatype EP should be contained in K_0, the constructor *EP* in Γ_0, and a suitable representation and embedding-projection pair for EP should be contained in Ψ_0 and E_0, respectively. Even though it sounds strange, there exists an embedding-projection pair for the EP datatype. The structural representation can be abbreviated as

$$\mathsf{Str}(\mathrm{EP})\ A_1\ A_2 \equiv \mathrm{Prod}\ (A_1 \to A_2)\ (A_2 \to A_1)\ ,$$

and the embedding-projection pair can be defined as follows:

$$\begin{aligned}\mathsf{ep}(\mathrm{EP}) = \mathbf{let}\ & x_{\mathrm{from}} = \lambda y \to \mathbf{case}\ y\ \mathbf{of}\ EP\ x_1\ x_2 \to x_1 \times x_2\\ & x_{\mathrm{to}} \ \ \ = \lambda y \to \mathbf{case}\ y\ \mathbf{of}\ x_1 \times x_2 \ \ \to EP\ x_1\ x_2\\ \mathbf{in}\ \ & EP\ x_{\mathrm{from}}\ x_{\mathrm{to}}\ .\end{aligned}$$

The datatype declarations D_i are checked for well-formedness using the rule (p-data) from Chapter 3 which still applies unchanged. Datatypes need not be translated. They appear in the resulting FCR program as they appear in the FCR+gf program, without any modifications. The datatypes are checked under the environment K, which in addition to K_0 contains all kinds of the datatypes defined in the program. This means that datatypes can be mutually recursive.

The environment K is also used to check and translate the main expression of the program, e. The other environments that are used in this position are Γ, containing Γ_0 plus all constructors defined by the D_i, and Ψ, containing Ψ_0 and all structural type representations of the D_i. The expression e is translated with an empty dependency environment and an empty signature environment. There is no need to pass the E environment around while checking and translating the expression. We only require the information which datatypes have embedding-projection pairs, and we can query Ψ for that, since we assume that both Ψ and E have associated entries.

Not plain e is translated, though, but rather

$$\mathbf{let}\ d_{\mathrm{from}}; d_{\mathrm{to}}\ \mathbf{in}\ \mathbf{let}\ d_{\mathrm{bimap}}\ \mathbf{in}\ e\ .$$

The declarations in the outer let are simply the *from* and *to* functions that we have already used before

$$\begin{aligned} d_{\mathrm{from}} &\equiv \mathit{from} = \lambda x \to \mathbf{case}\ x\ \mathbf{of}\ EP\ \mathit{from}\ \mathit{to} \to \mathit{from}\\ d_{\mathrm{to}} &\equiv \mathit{to} \ \ \ \ \ = \lambda x \to \mathbf{case}\ x\ \mathbf{of}\ EP\ \mathit{from}\ \mathit{to} \to \mathit{to}\ ,\end{aligned}$$

and d_{bimap} contains a definition of the generic function *bimap*, much as the one given in Section 11.2, but desugared (i.e., written as a typecase, *id* rewritten to $\lambda x \to x$, and composition expanded to application). We assume the d_{bimap} contains cases for the primitive abstract types in K_0, such as Int, Float and Char.

Due to the presence of these let bindings, not only the programmer, but also the translation can assume the presence of the *bimap* function.

The $[\![\cdot]\!]^{\mathrm{gf}}$ translation on expressions is mostly the same as the $[\![\cdot]\!]^{\mathrm{par}}$ translation from Chapters 6, using the revised generic application algorithm from Chapter 9 – with the exception that $[\![\cdot]\!]^{\mathrm{gf}}$ additionally passes around the Ψ environment. The one significant difference is in how we translate declarations of type-indexed

$$[\![d_{\text{FCR+tif+par}} \rightsquigarrow \Gamma_2 ; \Sigma_2]\!]^{\text{gf}}_{K;\Gamma_1;\Delta;\Sigma_1;\Psi} \equiv \{d_{\text{FCR}\ i}\}^{i \in 1..n}_{;}$$

$$\frac{\begin{array}{c} P_0 \equiv T\ \{\alpha_k\}^{k\in 1..\ell} \\ \mathsf{Str}(T) \equiv \{\Lambda a_k :: \kappa_k .\}^{k\in 1..\ell}\ t \in \Psi \\ K;\Gamma' \vdash^{tpsig} x\ \langle \pi \rangle :: \sigma \\ \pi \equiv (\{b_i :: *\}^{i\in 1..r} \mid \{b'_j :: \kappa'_j\}^{j\in 1..s}) \\ \{\beta_i \text{ fresh}\}^{i\in 1..r} \qquad \{\beta'_j \text{ fresh}\}^{j\in 1..s} \\ \mathsf{deptt}_{K;\Gamma}(x\ \langle\pi\rangle, x) \equiv \pi \\ K' \equiv K\ \{,\beta_i :: *\}^{i\in 1..r}\ \{,\beta'_j :: \kappa'_j\}^{j\in 1..s} \\ e_0 \equiv \textbf{let}\ \{bimap\ \langle\beta_i\rangle = \mathsf{ep}(T)\}^{i\in 1..r}_{;} \\ \quad \{bimap\ \langle\beta'_j\rangle = EP\ id\ id\}^{j\in 1..s}_{;} \\ \textbf{in}\ \ to\ bimap\ \langle \mathsf{mbase}_{K';\Gamma}(x\ \langle \{\beta_i\}^{i\in 1..r}_{,} \mid \{\beta'_j\}^{j\in 1..s}_{,}\rangle)\rangle \\ (x\ \langle t\{[\alpha_k\ /\ a_k]\}^{k\in 1..\ell}\rangle) \\ d' \equiv x\ \langle a\rangle = \textbf{typecase}\ a\ \textbf{of}\ P_0 \to e_0\ \{; P_i \to e_i\}^{i\in 1..n} \\ [\![d' \rightsquigarrow \Gamma';\Sigma']\!]^{\text{gf}}_{K;\Gamma;\Delta;\Sigma;\Psi} \equiv \{d_{\text{FCR}\ i}\}^{i\in 1..n}_{;} \end{array}}{\begin{array}{r} [\![x\ \langle a\rangle = \textbf{typecase}\ a\ \textbf{of}\ \{P_i \to e_i\}^{i\in 1..n}_{;} \rightsquigarrow \Gamma';\Sigma']\!]^{\text{gf}}_{K;\Gamma;\Delta;\Sigma;\Psi} \\ \equiv \{d_{\text{FCR}\ i}\}^{i\in 1..n}_{;} \end{array}} \quad \text{(d/tr-gf)}$$

Figure 11.2: Translation of FCR+gf declarations to FCR, extends Figure 6.15

functions: we define a companion for the rule (d/tr-tif) from Figure 6.15, named rule (d/tr-gf) in Figure 11.2. This rule can be applied *before* rule (d/tr-tif), to add arms for additional datatypes to a generic function.

If x is defined via typecase on patterns $\{P_i\}_{,}^{i \in 1..n}$, we can use rule (d/tr-gf) to derive generically an additional case for pattern P_0. We implicitly assume that the named type T appearing in P_0 is not yet covered by any of the other type patterns. We need the structural representation of T out of environment Ψ, which is of the form $\{\Lambda a_k :: \kappa_k .\}^{k \in 1..\ell}\ t$. We are going to extend the typecase of x with a new arm $P_0 \to e_0$. As discussed before, e_0 is in essence a generic application of x to the structural representation type, of the form $x\ \langle t\{[\alpha_k\ /\ a_k]\}^{k \in 1..\ell}\rangle$, where the formal parameters a_k of the structural representation type are replaced by the formal dependency parameters α_k in pattern P_0. The call of x is surrounded by a call to *bimap*, applied to the base type of x, to lift the isomorphism between T and $\mathsf{Str}(T)$ to the type of x. In the base type we replace all generic and non-generic slots with fresh dependency variables, and use local redefinition to plug in $\mathsf{ep}(T)$ for the generic slots, and *EP id id* for the non-generic slots.

The extended typecase is bound to d'. We return the translation of d' as a result, where d' is translated using the same judgment recursively. After adding this one new case, we can thus either add more cases, for more types, or ultimately fall back to rule (d/tr-tif) to translate all cases – the user-defined original ones plus the newly generated generic ones – in the same way as for any type-indexed function.

The condition $\mathsf{deptt}_{K;\Gamma}(x\ \langle \pi \rangle, x) \equiv \pi$ implements the reflexivity condition that we have pointed out in Section 11.4. The function x must depend on itself, and the dependency must use the same type tuple that appears on the left hand side of the type signature.

To see why this additional requirement is necessary, consider the functions x and y with type signatures as follows:

$$\begin{array}{lll} x\ \langle a_1, a_2 \rangle & :: (x\ \langle a_1, a_1 \rangle, y\ \langle a_2 \rangle) & \Rightarrow \text{Either}\ a_1\ a_2 \\ y\ \langle a \rangle & :: (y\ \langle a \rangle) & \Rightarrow a\ . \end{array}$$

Here, the dependency of x on itself is restricted, because $\mathsf{deptt}(x\ \langle a_1, a_2 \rangle, x) \equiv a_1, a_1$. We then have

$$\begin{array}{l} x\ \langle \text{Id}\ \alpha \rangle :: \forall (a_1 :: *)\ (a_2 :: *).\ (x\ \langle \alpha \rangle :: \text{Either}\ a_1\ a_1, y\ \langle \alpha \rangle :: a_2) \\ \qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \text{Either}\ (\text{Id}\ a_1)\ (\text{Id}\ a_2)\ . \end{array}$$

If we use rule (d/tr-gf) to generate an arm for type Id, we end up with

$$\begin{array}{l} x\ \langle \text{Id}\ \alpha \rangle = \textbf{let}\ \mathit{bimap}\ \langle \beta_1 \rangle = \mathsf{ep}(\text{Id}) \\ \qquad\qquad\qquad\quad \mathit{bimap}\ \langle \beta_2 \rangle = \mathsf{ep}(\text{Id}) \\ \qquad\qquad\quad\ \textbf{in}\ \ \mathit{to}\ \mathit{bimap}\ \langle \text{Either}\ \beta_1\ \beta_2 \rangle\ x\ \langle \alpha \rangle\ , \end{array}$$

because $\mathsf{Str}(\mathrm{Id})\ \alpha \equiv \alpha$. The right hand side is of type

$$\begin{aligned}\forall(a_1 :: *)\ (a_2 :: *).\ (x\ \langle\alpha\rangle :: \mathrm{Either}\ a_1\ a_2, y\ \langle\alpha\rangle :: a_2)\\ \Rightarrow \mathrm{Either}\ (\mathrm{Id}\ a_1)\ (\mathrm{Id}\ a_1)\ ,\end{aligned}$$

which does not match the type above that is required for $x\ \langle\mathrm{Id}\ \alpha\rangle$.

11.6 How to determine the required components

Translation via rule (d/tr-gf) in Figure 11.2 introduces nondeterminism into the translation, because it is not exactly specified which additional components are generated, and in which order. Now, the order is irrelevant, because the new arms do not depend on the already existing arms, and the order of arms in a typecase is irrelevant. Furthermore, the rule ensures that a consistent set of additional components is generated: the generation of additional components extends the signature environment, but all components that are referred to in the body (i.e., including the additional components), must be present.

But the rule does not specify that no extraneous components must be generated (and, in fact, it does not do any harm to do so – they will end up as unused declarations in a let, thus be dead code). Maybe more importantly, it does not give any hints as to how an actual implementation could proceed to find out which specializations are needed.

First of all, it is reassuring to note that the number of required additional components will always be finite, simply because all components for x are of the form $x\ \langle T\rangle$, where T is a named type (and not a type expression). Of named types there are the ones declared in the beginning of the program, plus types contained in the initial kind environment K_0. Provided that K_0 is finite, the number of named types in finite. Even if it is infinite (because, for example, K_0 might contain all tuple types), only finitely many of them can ever be used in one program.

Therefore, there is the conservative option to generate components for any combination of type-indexed function and datatype that is used in the program, whenever possible. A function cannot be specialized to a certain datatype if the function is not sufficiently generic (some of the arms for Unit, Prod, Sum, and Zero are missing, or the function is not reflexive), if the datatype is abstract and there is no explicit arm, or if the datatype is defined in terms of a datatype to which the function cannot be specialized.

However, we can do much better than that and generate only as many components as needed to produce a consistent program. To this end, we simply let the translation drive the generation of additional components. Whenever we specialize a generic application that requires a certain component of a type-indexed function to exist, we try to produce this component. The generation of a new component and its translation might refer to yet other components and may thus recursively cause new components to be generated. As long as all these generations succeed, everything is fine. When one of the generations is not possible, we report a specialization error. But we never generate a component that is not required somewhere during the translation process. This process is guaranteed to yield the smallest set of components that produces a correct program, and it is guaranteed to terminate, because the "conservative" approach outlined above is the worst case.

11.7 Discussion of translation by specialization

Translation by specialization has one central advantage: the type arguments are eliminated from the code, and specialized instances are used on the different types. For a non-generic type-indexed function, there is no performance overhead, because at compile time, the specialized instances that are required at a call site are determined and filled in.

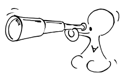
Furthermore, the translation itself is reasonably simple and efficient. As we have seen, we can determine the set of additional components that is needed for generic functions easily by essentially scanning the translated code for references to components that do not yet exist and then providing them. One very important point is that the specialization of calls to generic functions is compositional: components are always generated for a combination of a type-indexed function and a named type; calls to type expressions can be specialized in a way that the resulting expression is constructed from such basic components. This is absolutely essential to our approach, and enables to have an always terminating specialization algorithm, also for nested types or generic functions involving weird forms of recursion.

A disadvantage of specialization is the fact that type-indexed functions are not first class. In our language, the name of a type-indexed function on its own is not a legal expression – it may *exclusively* appear in the context of a generic application. If first class generic functions are allowed, it becomes harder to determine which components are required and where they have to be used. Allowing a more liberal use of type-indexed functions is the subject of ongoing and future work.

Another disadvantage of the specialization approach is that the code for the derived components of generic functions is not at all efficient: values are converted to isomorphic structural representations and back at runtime, everytime a generic function is called, and possibly several times if the function is recursive – and the vast majority of generic functions is recursive.

To obtain an efficient program, it is therefore crucial to perform optimizations on the generated code, where we can utilize the knowledge that we are indeed only performing conversions of values between two isomorphic representations. We have therefore implemented an experimental post-optimizer for the Generic Haskell compiler that performs partial evaluation on the generated components, thereby aggressively inlining and reducing all the conversions (de Vries 2004). The generic programming extension in Clean makes use of a similar optimization step (Alimarine and Smetsers 2004). These approaches show promising results: a few programs are still difficult to optimize, but for many examples, the resulting code after optimization is very much like what one would have written by hand as a specialized component for the generic function.

11.8 Other translation techniques



Translation by specialization is not the only possibility to handle type-indexed functions. In this section we sketch two alternative techniques, and evaluate their differences with the specialization approach.

11.8.1 Explicit type arguments and equality types

Cheney and Hinze (2002) developed a method of generic programming within Haskell, using already available extensions. An important idea is that type equality can be expressed in Haskell as a datatype:

$$\textbf{data}\ \mathrm{Equal}\ a\ b = \mathit{Proof}\ (\forall f :: * \to *. f\ a \to f\ b)\ .$$

The only possible value for this datatype – undefined aside – is the identity function, therefore the existence of a value of type Equal $a\ b$ constitutes a proof that a and b are the same type. It is possible to derive coercion functions of types $a \to b$ and $b \to a$ from such a proof. This equality type has independently been proposed by Baars and Swierstra (2002) in order to implement dynamic typing.

Cheney and Hinze describe an application of equality types to generic programming: a datatype of type representations is defined using equality types, and type analysis can then proceed as an ordinary **case** construct. Generic functions written in this style have to be augmented with several applications of coercion functions that employ the proofs from the type representations to convince the type system that certain types are equal. Furthermore, the automatic translation of datatypes to their structural representations still requires manual intervention by the programmer.

If direct support for equality types would be added to the compiler (Hinze 2003; Cheney and Hinze 2003), the type coercions could be inferred automatically.

However, additional annotations are not as bad if the encoding is used as a target language for Generic Haskell: both type coercions and the structural representation could be automatically generated.

The examples of generic functions that are discussed in the mentioned papers are less expressive than what we need to handle the generic functions in this thesis. In Section 6.6 we have explained that we translate generic functions into functions that have a kind-indexed type. Interestingly, the approach can be adapted to simulate kind-indexed types, following a joint idea of the author of this thesis and Ralf Hinze: the datatype of type representations encodes the type of the type-indexed function.

```
data Generic (f :: ∗ → ∗) (r :: ∗) =
                           Unit′ (Equal r               (f Unit))
   | ∀(a :: ∗) (b :: ∗). Sum′  (Equal r (f a → f b → f (Sum a b))
   | ∀(a :: ∗) (b :: ∗). Prod′ (Equal r (f a → f b → f (Prod a b))
   |                       Var  r
   | ∀(a :: ∗) (b :: ∗). App (Generic f (a → b)) (Generic f a) (Equal r b)
   | ∀(a :: ∗) (b :: ∗). Lam (Generic f a → Generic f b)        (Equal r (a → b))
```

The type Generic (which makes use of "existentially" quantified type variables in the syntax used by GHC) can be used in the type of a generic function, for example *enum*:

```
type Enum′ (a :: ∗) = [a]
type Enum (r :: ∗)  = Generic Enum′ r
enum :: ∀r :: ∗. Enum r → r
```

The type Generic takes two type arguments. The first one, for which we pass Enum′, is the type of the type-indexed function, the second is used to determine the type *resulting* from an application of the type-indexed function. The function *enum* takes a type representation as argument, and this type representation determines the type of the result. If we assume that

```
refl :: ∀a :: ∗. Equal a a
refl = Proof id
```

and

```
unit  = Unit′  refl
sum   = Sum′   refl
prod  = Prod′  refl
```

```
app x₁ x₂ = App x₁ x₂ refl
lam x     = Lam x     refl ,
```

we have for instance that

$$enum\ (app\ (app\ sum\ unit)\ unit) :: [\text{Sum Unit Unit}]\ .$$

The advantage of this translation of type-indexed functions is that in principle, the result is a first-class generic function. It is still parametrized by a type argument (encoded as a representation), and at the same time, it is an ordinary Haskell function. First-class generic functions are desirable, because they allow to parametrize generic functions with other generic functions.

The situation is, however, a bit more difficult: the type Generic as defined above can only hold generic functions with signature $\{\text{Unit}, \text{Sum}, \text{Prod}\}$. If a function needs an explicit definition for Int, the type Generic must be extended with a case for Int:

```
| Int (Equal r (f Int)) .
```

But type-indexed functions with different signatures require different datatypes, and are therefore incompatible, even if they have otherwise the same type. Nevertheless, we hope to pursue this approach further in the future and find a way to allow first-class generic functions in Generic Haskell.

11.8.2 Type class encoding

Another idea for translating Generic Haskell is by encoding Dependency-style Generic Haskell as directly as possible in the Haskell type class system, making use of multi-parameter type classes with functional dependencies (Jones 2000). Indeed, a type-indexed function such as *enum* can be encoded as a type class as follows:

```
class Enum (n :: *) (a :: *) | n → a where
  enum :: n → [a]
```

We use the additional type parameter *n* to represent the type argument of the generic function. For each case in the definition of *enum*, one instance is defined. For example, for Unit and Sum:

```
instance    Enum DUnit       Unit where
  enum DUnit =         [Unit]
```

```
instance    (Enum α            a,
             Enum β            b)
          ⇒  Enum (DSum α β) (Sum a b) where
  enum (DSum α β) = interleave (map Inl (enum α))
                               (map Inr (enum β))
```

The fact that *enum* depends on itself is reflected in the constraints on the instance declaration for Sum. A function that depends on more than one function would need additional constraints on the instance declarations. We use α and β as normal type variables here, because they play the role of dependency variables. The types DUnit and DSum are used as *names* of the type Unit and Sum, which can be used both on the type and on the value level. Their definition is

```
data DUnit     = DUnit
data DSum a b = DSum a b
```

We now have

```
enum (DSum DUnit DUnit) :: [Sum Unit Unit] ,
```

and the expression evaluates to [*Inl* Unit, *Inr* Unit].

Local redefinition can be modelled by a local datatype and instance declaration. Assuming a similar encoding for the generic function *equal*, we could encode our example from Chapter 8,

```
let equal ⟨α⟩ = equalCaseInsensitive
    enum ⟨α⟩ = enum ⟨Char⟩
in  equal ⟨[α]⟩ "laMBdA" "Lambda" ,
```

as

```
let data Dα = Dα
    instance Equal Dα Char where
      equal Dα = equalCaseInsensitive
    instance Enum Dα Char where
      enum Dα = enum DChar
in  equal (DList Dα) "laMBdA" "Lambda" .
```

Of course, Haskell does not allow local datatypes, nor local instances. But we can lift both to the toplevel if we store the class methods in the datatype Dα:

```
data Dα a = Dα (a → a → Bool) [a]
instance Equal (Dα a) a where
  equal (Dα equalα enumα) = equalα
instance Enum (Dα a) a where
  enum (Dα equalα enumα) = enumα
let dα     = Dα equalα enumα
    equalα = equalCaseInsensitive
    enumα  = enum DChar
in  equal (DList dα) "laMBdA" "Lambda" .
```

The idea to simulate named instances (Kahl and Scheffczyk 2001) via functional dependencies, using datatypes as names, has been discussed in the Haskell community and been brought to the author's attention by Peter Thiemann.

Encoding Generic Haskell with the help of type classes is interesting for several reasons: first, it makes the relation between type classes and generic programming more obvious, and shows which features (such as local instances and datatypes) are currently lacking from the class system to support generic programming in the style of Generic Haskell with ease. Second, by reducing Generic Haskell programs to type class programs, we can make use of results regarding type inference for type classes. For instance, we can define variants of *equal* (or other type-indexed functions) that can be used without passing a type argument, by first defining

```
class DefaultInstance n a | a → n, n → a where
  di :: n
```

with instances such as

```
instance   DefaultInstance DUnit      Unit      where
  di = DUnit
instance   (DefaultInstance m a, DefaultInstance n b)
         ⇒ DefaultInstance (DSum m n) (Sum a b) where
  di = DSum di di  .
```

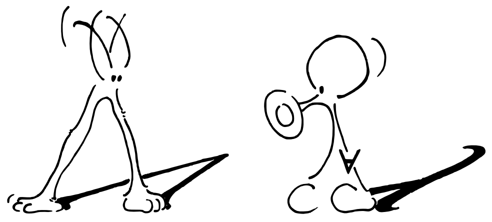
After that, we can define

```
class Equal′ a where
  equal′ :: a → a → Bool
instance (DefaultInstance n a, Equal n a) ⇒ Equal′ n a where
  equal′ = equal (di :: n) ,
```

and use *equal′* to let the compiler infer the type argument for *equal*. This solves the inference problem for type arguments as discussed in Section 13.1.

A third reason for a class encoding is that it allows us to easily combine ordinary Haskell programs with generic programs. Generic Haskell could be used as a compiler on parts of a program, producing a Haskell library which can then be used by other programmers using an ordinary Haskell compiler that need not be aware of generic functions. Such an approach has been realized for PolyP by Norell and Jansson (2004). In the same paper, an application to Generic Haskell based on kind-indexed types (cf. Section 6.6) has also been attempted, but the Haskell class system turned out to be not very suitable to model functions with kind-indexed types. The Dependency-style view seems to help here.

12 Generic Abstraction



In this chapter, we add a new possibility to define type-indexed functions. So far, generic functions are functions defined by means of a **typecase** construct. There is no other way to define a type-indexed function. Every function that makes use of another generic function must either itself be defined via **typecase**, or be at most parametrically polymorphic.

Generic abstraction has been used by Hinze since he started to work on generic programming in Haskell (Hinze 1999*a*), to define type-indexed functions in terms of other type-indexed functions, in a mostly informal way. The term "generic abstraction" was coined in the "Generic Haskell, specifically" paper (Clarke and Löh 2003), where this technique of defining new functions is one of the extensions described. The description in that paper, although clearly defined, makes use of a very naive translation that can lead to non-terminating specializations in some cases. The implementation that is presented in this chapter does not suffer from this limitation and integrates smoothly with the dependency type system (Löh *et al.* 2003).

We motivate the use of generic abstractions in Section 12.1. Then we present some further examples in Sections 12.2 and 12.3, before we discuss the implementation of generic abstractions in Section 12.4. We discuss how to simulate type-indexed functions with type indices of higher kinds using generic abstractions in 12.5, and how to generalize to generic abstractions over multiple type arguments in Section 12.6.

12.1 Motivation

Let us recall one of the examples we used to demonstrate the power of local redefinition: the function *size*, defined in Section 8.2. If applied to a constant type argument, *size* always returns 0. But using local redefinition, we can compute the size of a data structure, for example the length of a list:

$$\bigl(\mathbf{let}\ \mathit{size}\ \langle\alpha :: *\rangle = \mathit{const}\ 1\ \mathbf{in}\ \mathit{size}\ \langle[\alpha]\rangle\bigr)\ [1,2,3,4,5]$$

evaluates to 5.

Likewise, we can count the number of labels in a tree: the expression

$$\bigl(\mathbf{let}\ \mathit{size}\ \langle\alpha :: *\rangle = \mathit{const}\ 1\ \mathbf{in}\ \mathit{size}\ \langle\mathrm{Tree}\ \alpha\rangle\bigr)\ (\mathit{Node}\ \mathit{Leaf}\ \texttt{'x'}\ \mathit{Leaf})$$

evaluates to 1.

If we need the sizes of lists and trees more often, it is also possible to capture the computation in a function:

$$\begin{aligned}\mathit{sizeList}\ &= \mathbf{let}\ \mathit{size}\ \langle\alpha :: *\rangle = \mathit{const}\ 1\ \mathbf{in}\ \mathit{size}\ \langle[\alpha]\rangle\\ \mathit{sizeTree} &= \mathbf{let}\ \mathit{size}\ \langle\alpha :: *\rangle = \mathit{const}\ 1\ \mathbf{in}\ \mathit{size}\ \langle\mathrm{Tree}\ \alpha\rangle\ .\end{aligned}$$

These functions are polymorphic – their types are

$$\begin{aligned}\mathit{sizeList} &:: \forall a :: *.\ [a] &&\to \mathrm{Int}\\ \mathit{sizeTree} &:: \forall a :: *.\ \mathrm{Tree}\ a &&\to \mathrm{Int}\ .\end{aligned}$$

They are, however, no longer type-indexed. The type argument of the generic function *size* has irrevocably been instantiated, once to $[\alpha]$, once to Tree α, and what remains is an ordinary function. In fact, we can write

$$\mathbf{let}\ \mathit{size}\ \langle\alpha :: *\rangle = \mathit{const}\ 1\ \mathbf{in}\ \mathit{size}\ \langle f\ \alpha\rangle$$

for *any* functor f of kind $* \to *$, but there is no way to capture this expression as a function, without binding f to one concrete type at the same time.

Generic abstraction lifts this limitation. We can define

$$\mathit{fsize}\ \langle\gamma :: * \to *\rangle = \mathbf{let}\ \mathit{size}\ \langle\alpha :: *\rangle = \mathit{const}\ 1\ \mathbf{in}\ \mathit{size}\ \langle\gamma\ \alpha\rangle\ ,$$

and we can then use *fsize* $\langle[\,]\rangle$ and *fsize* $\langle\mathrm{Tree}\rangle$ instead of being forced to define specialized versions such as *sizeList* and *sizeTree*. The feature is called **generic abstraction** because it allows us to abstract from the type argument represented by γ in the definition of *fsize*. Using generic abstraction, we can thus define functions that depend on a type argument, but are not defined using a **typecase**. Note that *fsize* cannot be parametrically polymorphic in its type argument γ, because *size* is not parametrically polymorphic in its type argument.

A speciality in the definition of *fsize* immediately catches the eye: the type argument between the type parentheses $\langle\cdot\rangle$ is of kind $* \to *$ here. The type argument used in generic abstractions is not restricted to kind $*$. Functions defined via generic abstraction are thus different from ordinary type-indexed functions in some respects, but of course, they also have type signatures. This is the type signature for *fsize*:

$$\mathit{fsize}\ \langle f :: * \to * \mid a :: *\rangle :: (\mathit{size}\ \langle f\rangle) \Rightarrow f\ a \to \mathrm{Int}\ .$$

The function has one generic type parameter f of kind $* \to *$, and one non-generic type parameter a of kind $*$, because *fsize* is polymorphic in the element type of the data structure. There is a dependency on *size*, but not on itself. Functions defined by generic abstraction *never* depend on themselves. Only type-indexed functions defined by a typecase occur in dependency constraint sets. We will discuss the type signatures of functions defined by generic abstraction in Section 12.4.

Another typical example for the use of generic abstraction arises if we want to generically define *zip* in terms of *zipWith* (cf. page 136):

$$\begin{aligned}
&\mathit{zip}\ \langle f :: * \to * \mid a :: *, b :: *\rangle && :: (\mathit{zipWith}\ \langle f, f, f\rangle) \Rightarrow f\ a \to f\ b \to f\ (a, b)\\
&\mathit{zip}\ \langle \gamma :: * \to *\rangle && = \mathit{zipWith}\ \langle\gamma\rangle\ (,)\ .
\end{aligned}$$

Here, we have made use of the *short notation* (defined in Section 8.3) for *zipWith*. The function *zipWith* had a type of arity $\langle 3 \mid 0\rangle$, hence the three type variables in the dependency.

The use of generic abstraction is not confined to type arguments of kind $* \to *$. An example of a kind $*$ generic abstraction is the function *card*, which determines the cardinality of a datatype:

$$\begin{aligned}
&\mathit{card}\ \langle a :: *\rangle\ :: (\mathit{enum}\ \langle a\rangle) \Rightarrow \mathrm{Int}\\
&\mathit{card}\ \langle \alpha :: *\rangle = \mathit{fsize}\ \langle[\,]\rangle\ (\mathit{enum}\ \langle\alpha\rangle)\ .
\end{aligned}$$

We first enumerate the datatype in question, and then apply *fsize* to determine the number of elements in the resulting list (of course, we could have used the standard list function *length* just as well, but that would not serve as an example of generic abstraction). This function will only terminate for finite datatypes (we will define a variant of *card* that can detect infinite datatypes in Section 17.1.3). This example demonstrates that there is no problem in reusing one function that is defined by generic abstraction (*fsize*) in another. The dependency of *fsize* on *size* does not propagate to *card*, because *fsize* is used here on a constant type constructor.

Another example where we could already have used generic abstraction is the definition of *decodes*: the counterpart to *encode*, to retrieve a typed value from a list of bits, has type

$$\mathit{decodes}\ \langle a :: *\rangle :: (\mathit{decodes}\ \langle a\rangle) \Rightarrow [\mathrm{Bit}] \to [(a, [\mathrm{Bit}])]$$

The result type contains the possibility for failure (empty list), and can store a remaining part of the input. This is necessary for the recursive calls of *decodes*, but does not mirror the type of *encode*, which is

$$\mathit{encode}\ \langle a :: *\rangle\ :: (\mathit{encode}\ \langle a\rangle)\ \Rightarrow a \to [\mathrm{Bit}]\ .$$

Using, generic abstraction, we can now define a proper inverse for *encode*, named *decode*:

```
decode ⟨a :: ∗⟩  :: (decodes ⟨a⟩) ⇒ [Bit] → a
decode ⟨α :: ∗⟩ x = case decodes ⟨α⟩ x of
  [(y, [])] → y
  _         → error "no parse" .
```

For a value x of type t, we have

$$(\mathit{decode}\ \langle t\rangle \cdot \mathit{encode}\ \langle t\rangle)\ x \equiv x\ ,$$

provided there is no specialization error.

12.2 Generic reductions

A very large class of useful generic operations – including *size* and *collect* – can be expressed in terms of *reduce* via generic abstraction. The function *reduce* is defined normally, using a case analysis on the type argument, and takes a (supposedly) neutral element e and a binary operator $\oplus$ as arguments.

```
reduce ⟨a :: ∗ | c :: ∗⟩ :: (reduce ⟨a | c⟩ ⇒ c → (c → c → c) → a → c
reduce ⟨Int⟩     e (⊕) x    = e
reduce ⟨Char⟩    e (⊕) x    = e
```

$$\begin{aligned}
&\mathit{reduce}\ \langle\text{Float}\rangle && e\ (\oplus)\ x && = e\\
&\mathit{reduce}\ \langle\text{Unit}\rangle && e\ (\oplus)\ \mathit{Unit} && = e\\
&\mathit{reduce}\ \langle\text{Sum}\ \alpha\ \beta\rangle && e\ (\oplus)\ (\mathit{Inl}\ x) && = \mathit{reduce}\ \langle\alpha\rangle\ e\ (\oplus)\ x\\
&\mathit{reduce}\ \langle\text{Sum}\ \alpha\ \beta\rangle && e\ (\oplus)\ (\mathit{Inr}\ x) && = \mathit{reduce}\ \langle\beta\rangle\ e\ (\oplus)\ x\\
&\mathit{reduce}\ \langle\text{Prod}\ \alpha\ \beta\rangle && e\ (\oplus)\ (x \times y) && = \mathit{reduce}\ \langle\alpha\rangle\ e\ (\oplus)\ x \oplus \mathit{reduce}\ \langle\beta\rangle\ e\ (\oplus)\ y\ .
\end{aligned}$$

All constant types are replaced by e. We descend into sums in both directions, and combine the two partial results in a product using $\oplus$.

Using reduce, we can now define:

$$\begin{aligned}
&\mathit{collect}\ \langle a :: * \mid c :: *\rangle && :: (\mathit{reduce}\ \langle a \mid [c]\rangle) \Rightarrow a \to [c]\\
&\mathit{collect}\ \langle\alpha :: *\rangle && = \mathit{reduce}\ \langle\alpha\rangle\ [\,]\quad (+\!\!+)\\
&\mathit{size}\quad \langle a :: *\rangle && :: (\mathit{reduce}\ \langle a \mid \text{Int}\rangle) \Rightarrow a \to \text{Int}\\
&\mathit{size}\quad \langle\alpha :: *\rangle && = \mathit{reduce}\ \langle\alpha\rangle\ 0\quad (+)\\
&\mathit{conj}\quad \langle a :: *\rangle && :: (\mathit{reduce}\ \langle a \mid \text{Bool}\rangle) \Rightarrow a \to \text{Bool}\\
&\mathit{conj}\quad \langle\alpha :: *\rangle && = \mathit{reduce}\ \langle\alpha\rangle\ \mathit{True}\ \ (\wedge)\\
&\mathit{disj}\quad \langle a :: *\rangle && :: (\mathit{reduce}\ \langle a \mid \text{Bool}\rangle) \Rightarrow a \to \text{Bool}\\
&\mathit{disj}\quad \langle\alpha :: *\rangle && = \mathit{reduce}\ \langle\alpha\rangle\ \mathit{False}\ (\vee)
\end{aligned}$$

The thus defined functions *collect* and *size* are similar, but not identical to the stand-alone versions. They are still most valuable when used on a non-constant type argument, in the context of a local redefinition. But these versions all depend on *reduce* rather than on themselves! To compute the size of a list, one now needs to write

$$\textbf{let}\ \mathit{reduce}\ \langle\alpha\rangle\ e\ (\oplus)\ x = 1\ \textbf{in}\ \mathit{size}\ \langle[\alpha]\rangle\ .$$

The short notation that has been introduced in Section 8.3 can without any problems be extended slightly such that it works for any function that has exactly one dependency (instead of being restricted to functions that depend on only itself), and we can alternatively write

$$\mathit{size}\ \langle[\,]\rangle\ (\lambda e\ (\oplus)\ x \to 1)$$

to express the same: the dependency on *reduce* is supplied as a positional argument to *size*, using an anonymous function that ignores its three arguments and returns 1.

A reimplementation of *fsize* that works for any type constructor of kind $* \to *$ can be arranged via either of

$$\begin{aligned}
&\mathit{fsize}\ \langle\gamma :: * \to *\rangle = \mathit{reduce}\ \langle\gamma\rangle\ (\lambda e\ (\oplus)\ x \to 1)\ 0\ (+)\\
&\mathit{fsize}\ \langle\gamma :: * \to *\rangle = \mathit{size}\quad\ \langle\gamma\rangle\ (\lambda e\ (\oplus)\ x \to 1)\ ,
\end{aligned}$$

again employing short notation in both variants.

Similarly, the functions *conj* and *disj* defined above can be a foundation for the definition of generic versions of Haskell's list functions *and*, *or*, *any*, and *all*:

$$
\begin{array}{lll}
\mathit{and}\ \langle f :: * \to * \rangle & :: & (\mathit{reduce}\ \langle f \mid \mathrm{Bool} \rangle) \Rightarrow f\ \mathrm{Bool} \to \mathrm{Bool} \\
\mathit{and}\ \langle \gamma :: * \to * \rangle & = & \mathit{conj}\ \langle \gamma \rangle\ (\lambda e\ (\oplus)\ x \to x) \\
\mathit{or}\ \ \langle f :: * \to * \rangle & :: & (\mathit{reduce}\ \langle f \mid \mathrm{Bool} \rangle) \Rightarrow f\ \mathrm{Bool} \to \mathrm{Bool} \\
\mathit{or}\ \ \langle \gamma :: * \to * \rangle & = & \mathit{disj}\ \ \langle \gamma \rangle\ (\lambda e\ (\oplus)\ x \to x) \\
\mathit{all}\ \ \langle f :: * \to * \mid a :: * \rangle & :: & (\mathit{reduce}\ \langle f \mid \mathrm{Bool} \rangle) \Rightarrow (a \to \mathrm{Bool}) \to f\ a \to \mathrm{Bool} \\
\mathit{all}\ \ \langle \gamma :: * \to * \rangle\ p & = & \mathit{conj}\ \langle \gamma \rangle\ (\lambda e\ (\oplus)\ x \to p\ x) \\
\mathit{any}\ \langle f :: * \to * \mid a :: * \rangle & :: & (\mathit{reduce}\ \langle f \mid \mathrm{Bool} \rangle) \Rightarrow (a \to \mathrm{Bool}) \to f\ a \to \mathrm{Bool} \\
\mathit{any}\ \langle \gamma :: * \to * \rangle\ p & = & \mathit{disj}\ \ \langle \gamma \rangle\ (\lambda e\ (\oplus)\ x \to p\ x)\ \ .
\end{array}
$$

Yet more instances of *reduce* are the generalization of list concatenation *concat*, the flattening of a data structure into a list *flatten*, and a function *compose* that can compose a data structure containing functions into one function.

$$
\begin{array}{lll}
\mathit{concat}\ \ \langle f :: * \to * \mid a :: * \rangle & :: & (\mathit{reduce}\ \langle f \mid [a] \rangle) \Rightarrow f\ [a] \to [a] \\
\mathit{concat}\ \ \langle \gamma :: * \to * \rangle & = & \mathit{collect}\ \langle \gamma \rangle\ \mathit{id} \\
\mathit{flatten}\ \ \langle f :: * \to * \mid a :: * \rangle & :: & (\mathit{reduce}\ \langle f \mid [a] \rangle) \Rightarrow f\ a \to [a] \\
\mathit{flatten}\ \ \langle \gamma :: * \to * \rangle & = & \mathit{collect}\ \langle \gamma \rangle\ (\lambda x \to [x]) \\
\mathit{compose}\ \langle f :: * \to * \mid a :: * \rangle & :: & (\mathit{reduce}\ \langle f \mid a \to a \rangle) \Rightarrow f\ (a \to a) \to (a \to a) \\
\mathit{compose}\ \langle \gamma :: * \to * \rangle & = & \mathit{reduce}\ \langle \gamma \rangle\ (\lambda e\ (\oplus)\ x \to x)\ \mathit{id}\ (\cdot)\ \ .
\end{array}
$$

12.3 Cata- and anamorphisms

If we have a representation of a datatype with explicit fixpoints, using the fixpoint datatype

$$\mathbf{data}\ \mathrm{Fix}\ (a :: * \to *) = \mathit{In}\ (a\ (\mathrm{Fix}\ a))\ ,$$

we can express catamorphisms and anamorphisms – their list variants are better known as *fold* and *unfold* in the Haskell world – on that datatype using generic abstraction over the generic *map* function.

Assume that *out* is defined as follows:

$$\mathit{out}\ (\mathit{In}\ x) = x\ \ .$$

Then *cata* and *ana* are defined by:

$$
\begin{array}{l}
\mathit{cata}\ \langle f :: * \to * \mid a :: * \rangle\ ::\ (\mathit{map}\ \langle f, f \rangle) \Rightarrow (f\ a \to a) \to \mathrm{Fix}\, f \to a \\
\mathit{ana}\ \ \langle f :: * \to * \mid a :: * \rangle\ ::\ (\mathit{map}\ \langle f, f \rangle) \Rightarrow (a \to f\ a) \to a \to \mathrm{Fix}\, f
\end{array}
$$

$$\begin{aligned}
&\mathit{cata}\ \langle\gamma :: * \to *\rangle\ \mathit{alg} &&= \mathit{alg} \cdot \mathit{map}\ \langle\gamma\rangle\ (\mathit{cata}\ \langle\gamma\rangle\ \mathit{alg}) \cdot \mathit{out}\\
&\mathit{ana}\ \langle\gamma :: * \to *\rangle\ \mathit{coalg} &&= \mathit{In} \cdot \mathit{map}\ \langle\gamma\rangle\ (\mathit{ana}\ \langle\gamma\rangle\ \mathit{coalg}) \cdot \mathit{coalg}\ .
\end{aligned}$$

The occurrences of *cata* $\langle\gamma\rangle$ and *ana* $\langle\gamma\rangle$ on the right hand side do not trigger dependencies – they denote ordinary recursion. The function *cata* could as well have been implemented via

$$\mathit{cata}\ \langle\gamma :: * \to *\rangle = \mathbf{let}\ f\ \mathit{alg} = \mathit{alg} \cdot \mathit{map}\ \langle\gamma\rangle\ (f\ \mathit{alg}) \cdot \mathit{out}\ \mathbf{in}\ f$$

instead.

If we want to use these functions on a recursive datatype, we first have to define the datatype in terms of Fix. For example, the datatype of lists can isomorphically be defined as:

```
type List  (a :: *)          = Fix (ListF a)
data ListF (a :: *) (r :: *) = NilF | ConsF a r  .
```

The datatype ListF is also called the *pattern functor* of List (hence the "F" in the names).

The type of built-in lists is isomorphic to the type List. We can define an embedding-projection pair that holds isomorphisms:

```
epList :: ∀a :: *. EP (List a) [a]
epList = let fromList (In NilF)          = [ ]
             fromList (In (ConsF x r))   = x : fromList r
             toList   [ ]                = In NilF
             toList   (x : xs)           = In (ConsF x (toList xs))
         in  EP fromList toList  .
```

The ListF type is not recursive, but in turn, the conversion functions are. This is different from the standard view on datatypes discussed in Chapter 10, where the structural representation of a recursive datatype is not directly recursive, but still refers to the original datatype.

On the list datatype, viewed as a fixpoint, we can define all the classical catamorphisms. Many of those can be expressed as reductions (cf. Section 12.2), but there are some that cannot, for instance:

```
reverse :: ∀a :: *. (List a) → [a]
reverse = let reverseAlg NilF          = [ ]
              reverseAlg (ConsF x r)   = r ++ [x]
          in  cata ⟨ListF⟩ reverseAlg  .
```

```
sublists :: ∀a :: ∗. (List a) → [[a]]
sublists = let sublistsAlg NilF        = [ ]
               sublistsAlg (ConsF x r) = map ⟨[ ]⟩ (x:) r ++ r
           in  cata ⟨ListF⟩ sublistsAlg .
```

Using the embedding-projection pair, we can explicitly specify the relation with the prelude function *foldr*:

```
foldr          :: ∀(a :: ∗) (b :: ∗). (a → b → b) → b → [a] → b
foldr (⊕) e xs = let alg NilF        = e
                     alg (ConsF x r) = x ⊕ r
                 in  cata ⟨[ ]⟩ alg (to epList xs)
```

Hinze (1999*b*) presents catamorphisms and anamorphisms in the way just described, using an explicit representation of datatypes based on the Fix type constructor. In PolyP (Jansson and Jeuring 1997), catamorphisms can be represented as generic functions more directly, because generic functions are defined on functors of kind $* \to *$ in PolyP, and the computation of the pattern functor is performed automatically by the compiler. We discuss a similar approach in Section 17.2.

12.4 Types and translation of generic abstractions

Functions defined via generic abstraction have type signatures that are almost like type signatures of ordinary type-indexed functions. As a slight generalization, generic type variables are not restricted to kind $*$. We have seen several examples for functions where the generic arguments were of kind $* \to *$ instead, among them

```
and ⟨f :: ∗ → ∗⟩          :: (reduce ⟨f | Bool⟩) ⇒ f Bool → Bool
cata ⟨f :: ∗ → ∗ | a :: ∗⟩ :: (map ⟨f,f⟩) ⇒ (f a → a) → Fix f → a .
```

Obviously, the generic type variables may be used in the dependencies, even though both *reduce* and *map* – as ordinary type-indexed functions defined via a typecase – normally take generic variables of kind $*$. A generic abstraction itself never appears in dependency constraints.

In Figure 12.1, we present the syntax extensions necessary to cover generic abstractions in our language. We call the extended language FCR+gf+gabs. We introduce a new form of value declaration to define a function via generic abstraction. It is syntactically distinguished from local redefinition by the kind annotation in the type argument (we keep the kind annotation not only in the

Value declarations

$d \quad ::= \ \ldots$ everything from Figures 4.1 and 8.3

$\quad\quad \mid \ x \ \langle \alpha :: \kappa \rangle = e$ generic abstraction

Type tuple patterns

$\pi \quad ::= \{a_i :: \kappa\}_{,}^{i \in 1..r} \mid \{b_j :: \kappa_j\}_{,}^{j \in 1..s}$

type tuple pattern

Figure 12.1: Core language with generic abstraction FCR+gf+gabs, extends language FCR+gf in Figures 9.4, 8.3, 6.1, 4.1, and 3.1

type signature mainly for this reason, but also to emphasize that generic abstractions can be for type arguments of higher kind). Furthermore, we generalize the syntax of type tuple patterns to allow generic type variables to be of arbitrary kind. Note, though, that all generic type variables must still be of the *same* kind. The kind checking rules for type and type argument tuples in Figure 9.5 have been formulated in such a way that they can directly deal with the more general situation.

$$\mathrm{K}; \Gamma; \Sigma \vdash^{tpsig} x \ \langle \pi \rangle :: \sigma$$

$$\frac{\begin{array}{c}
\{\mathrm{K}; \Gamma; \Sigma \vdash^{tpsig} y_k \ \langle \pi_k \rangle :: (\{z_{k,i} \ \langle \vartheta_{k,i} \rangle\}_{,}^{i \in 1..m_k}) \Rightarrow t_k\}^{k \in 1..n} \\
\{y_k \ \langle \mathsf{Abs} \rangle \notin \Sigma\}^{k \in 1..n} \qquad \{\{z_{k,i} \in \{y_j\}_{,}^{j \in 1..n}\}^{i \in 1..m_k}\}^{k \in 1..n} \\
\mathrm{K}_* \equiv \mathrm{K} \ \{, a_i :: *\}^{i \in 1..r} \ \{, b_j :: \kappa_j\}^{j \in 1..s} \\
\mathrm{K}' \ \equiv \mathrm{K} \ \{, a_i :: \kappa\}^{i \in 1..r} \ \{, b_j :: \kappa_j\}^{j \in 1..s} \\
\{\{\{z_{k,i} \equiv y_j \Longrightarrow \mathrm{K}_*, \pi_k \vdash \vartheta_{k,i} \leqslant \vartheta_k\}^{j \in 1..n}\}^{i \in 1..m_k}\}^{k \in 1..n} \\
\mathrm{K}' \vdash t :: * \qquad \{\mathrm{K}_* \vdash \vartheta_k :: \pi_k\}^{k \in 1..n} \qquad \{\vdash \vartheta_k \downarrow\}^{k \in 1..n}
\end{array}}{\begin{array}{l}
\mathrm{K}; \Gamma; \Sigma \vdash^{tpsig} x \ \langle \{a_i :: \kappa\}_{,}^{i \in 1..r} \mid \{b_j :: \kappa_j\}_{,}^{j \in 1..s} \rangle \\
\qquad\qquad :: (\{y_k \ \langle \vartheta_k \rangle\}_{,}^{k \in 1..n}) \Rightarrow t)
\end{array}} \quad \text{(typesig-m)}$$

Figure 12.2: Well-formedness of type signatures for type-indexed functions including generic abstractions, replaces Figure 9.11

The rule (typesig-m) to check well-formedness of type signatures of type-indexed functions, needs to be adapted. The changed rule, which we still call (typesig-m), is shown in Figure 12.2. As it stands now, it works for type signatures of all type-indexed functions, whether they are defined using a typecase or generic abstraction. There are a couple of differences with the old rule: we require that functions defined via generic abstractions never appear in the dependencies of a type-indexed function. We use the statement $x \langle \mathsf{Abs} \rangle \notin \Sigma$ to express that x is not defined via generic abstraction, for reasons that will become clear soon. This condition requires that the judgment is extended to take the signature environment Σ as additional input.

In addition to that, we extend the kind environment K twice in the new rule: once, for K′, we add the generic and non-generic type variables with the kinds declared in the type tuple pattern (where we now allow any kind for the generic variables); and once, in K_*, we add the non-generic type variables with their declared kinds, but the generic type variables with kind $*$. We then use K′ to kind check the base type t, but K_* to kind check the type tuples that go with the dependencies.

The translation of generic abstractions is simple: a generic abstraction x is translated to a single component, which is called $\mathsf{cp}(x, \mathsf{Abs})$. The identifier Abs denotes a special type name that cannot occur in user-written programs, and we treat any occurrence of $x \langle A \rangle$ as if it were the call $x \langle \mathsf{Abs}\ A \rangle$. For a generic abstraction of form $x \langle \alpha :: \kappa \rangle = e$, we assume that Abs is of kind $\kappa \to * $ – this corresponds to the fact that we treat generic type variables as kind $*$ in the dependencies.

The rule to check and translate a generic abstraction is rule (d/tr-gabs) in Figure 12.3 and is analogous to the rule (d/tr-tif) for a type-indexed function, only that it is simpler because there is only one arm to check. There are two rules now to type check and translate generic applications, also shown in Figure 12.3. One, (e/tr-genapp), is just like the former, with the additional condition that $x \langle \mathsf{Abs} \rangle \notin \Sigma$, which serves to verify that x is not defined by generic abstraction. The other, (e/tr-genapp-gabs), is for generic abstractions and performs the conversion of the type argument A to Abs A.

The only additional modification we then need is in the mbase function, where we ignore occurrences of Abs in the head of the generic slots – this corresponds to the fact that we use generic type variables with their declared kind in the base type. The modified base type judgments are shown in Figure 12.4. The generic application algorithm gapp now falls back on the thus modified base type function, and can be used without modification. Neither is the translation algorithm $[\![\cdot]\!]^{\mathrm{gtrans}}$ in need of change: because Abs is syntactically a type name, its appearance as the head of a type argument for the generic abstraction x causes a reference to $\mathsf{cp}(x, \mathsf{Abs})$ to appear in the translation.

Let us conclude this section with a few examples. Recall the function *fsize* from Section 12.1, which computes the size of a container type of kind $* \to *$. With

$$[\![d_{\textsc{fcr}+\text{gf}+\text{gabs}} \rightsquigarrow \mathrm{K}_2; \Gamma_2; \Delta_2; \Sigma_2]\!]^{\text{gabs}}_{\mathrm{K}_1;\Gamma_1;\Delta_1;\Sigma_1} \equiv \{d_{\textsc{fcr}}\}^{i \in 1..n}_{;}$$

$$\frac{\begin{array}{c} \mathrm{K}; \Gamma \vdash^{tpsig} x\ \langle \pi \rangle :: \sigma \\ \mathrm{K}' \equiv \mathrm{K}, \alpha :: \kappa, \mathsf{Abs} :: \kappa \to * \qquad \Gamma' \equiv x\ \langle \pi \rangle :: \sigma \qquad \Sigma' \equiv x\ \langle \mathsf{Abs} \rangle \\ \mathsf{gapp}_{\mathrm{K}';\Gamma,\Gamma'}(x\ \langle \mathsf{Abs}\ \alpha \rangle) \equiv q \quad \Delta \vdash q' \leqslant q \rightsquigarrow e''[\bullet] \\ [\![e :: q']\!]^{\text{gabs}}_{\mathrm{K}';\Gamma;\Delta;\Sigma} \equiv e' \end{array}}{[\![x\ \langle \alpha :: \kappa \rangle = e \rightsquigarrow \varepsilon; \Gamma'; \varepsilon; \Sigma']\!]^{\text{gabs}}_{\mathrm{K};\Gamma;\Delta;\Sigma} \equiv \mathsf{cp}(x, \mathsf{Abs}) = e''[e']} \quad \text{(d/tr-gabs)}$$

$$[\![e_{\textsc{fcr}+\text{gf}+\text{gabs}} :: t_{\textsc{fcr}+\text{gf}+\text{gabs}}]\!]^{\text{gabs}}_{\mathrm{K};\Gamma;\Delta;\Sigma} \equiv e_{\textsc{fcr}}$$

$$\frac{\begin{array}{c} x\ \langle \mathsf{Abs} \rangle \notin \Sigma \\ \mathrm{K} \vdash A :: * \\ \mathsf{gapp}_{\mathrm{K};\Gamma}(x\ \langle A \rangle) \equiv q \qquad [\![x\ \langle A \rangle]\!]^{\text{gtrans}}_{\mathrm{K};\Gamma;\Sigma} \equiv e \\ \mathrm{K}; \Delta \vdash q \leqslant t \end{array}}{[\![x\ \langle A \rangle :: t]\!]^{\text{gabs}}_{\mathrm{K};\Gamma;\Delta;\Sigma} \equiv e} \quad \text{(e/tr-genapp)}$$

$$\frac{\begin{array}{c} x\ \langle \mathsf{Abs} \rangle \in \Sigma \\ \mathrm{K}; \Gamma \vdash^{tpsig} x\ \langle \{a_i :: \kappa\}^{i \in 1..r}_{,} \mid \{b_j :: \kappa_j\}^{j \in 1..r}_{,} \rangle :: \sigma \\ \mathrm{K} \vdash A :: \kappa \qquad \mathrm{K}' \equiv \mathrm{K}, \mathsf{Abs} :: \kappa \to * \\ \mathsf{gapp}_{\mathrm{K};\Gamma}(x\ \langle \mathsf{Abs}\ A \rangle) \equiv q \qquad [\![x\ \langle \mathsf{Abs}\ A \rangle]\!]^{\text{gtrans}}_{\mathrm{K}';\Gamma;\Sigma} \equiv e \\ \mathrm{K}; \Delta \vdash q \leqslant t \end{array}}{[\![x\ \langle A \rangle :: t]\!]^{\text{gabs}}_{\mathrm{K};\Gamma;\Delta;\Sigma} \equiv e} \quad \text{(e/tr-genapp-gabs)}$$

Figure 12.3: Translation of generic abstractions to FCR

$$\mathsf{mbase}_{K;\Gamma}(x\ \langle\Theta\rangle) \equiv t$$

$$\frac{\begin{array}{c}\Theta \equiv \{A_i\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s} \qquad \{\mathsf{head}(A_i) \not\equiv \mathsf{Abs}\}^{i\in 1..r}\\ x\ \langle\pi\rangle :: (\{y_k\ \langle\vartheta_k\rangle\}_{,}^{k\in 1..n}) \Rightarrow t \in \Gamma\\ K \vdash^{pat} \Theta :: \pi \rightsquigarrow \varphi\end{array}}{\mathsf{mbase}_{K;\Gamma}(x\ \langle\Theta\rangle) \equiv \varphi\ t} \quad \text{(base-m)}$$

$$\frac{\begin{array}{c}\Theta \equiv \{\mathsf{Abs}\ A_i\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s} \qquad \Theta' \equiv \{A_i\}_{,}^{i\in 1..r} \mid \{t_j\}_{,}^{j\in 1..s}\\ x\ \langle\pi\rangle :: (\{y_k\ \langle\vartheta_k\rangle\}_{,}^{k\in 1..n}) \Rightarrow t \in \Gamma\\ K \vdash^{pat} \Theta' :: \pi \rightsquigarrow \varphi\end{array}}{\mathsf{mbase}_{K;\Gamma}(x\ \langle\Theta\rangle) \equiv \varphi\ t} \quad \text{(base-m-gabs)}$$

Figure 12.4: Modified base type judgment for FCR+gf+gabs, replaces Figure 9.9

type signature, the function can be defined as follows:

$$\begin{array}{lll} \mathit{fsize}\ \langle f :: * \to * \mid a :: *\rangle & :: & (\mathit{size}\ \langle f\rangle) \Rightarrow f\ a \to \mathrm{Int}\ .\\ \mathit{fsize}\ \langle\gamma :: * \to *\rangle & = & \mathbf{let}\ \mathit{size}\ \langle\alpha :: *\rangle = \mathit{const}\ 1\\ & & \mathbf{in}\ \ \mathit{size}\ \langle\gamma\ \alpha\rangle\ \ . \end{array}$$

We have one generic slot f, of kind $* \to *$, and one non-generic variable a of kind $*$, which represents the element type of the underlying data structure, in which *fsize* is polymorphic. The translation of the function, according to the rules given above, is

$$\begin{array}{lll} \mathsf{cp}(\mathit{fsize}, \mathsf{Abs})\ \mathsf{cp}(\mathit{fsize}, \gamma) & = & \mathbf{let}\ \mathsf{cp}(\mathit{size}, \alpha)\ \ = \mathit{const}\ 1\\ & & \mathbf{in}\ \ (\mathsf{cp}(\mathit{size}, \gamma))\ (\mathsf{cp}(\mathit{size}, \alpha))\ \ . \end{array}$$

If the function *fsize* is called on a type argument containing dependency variables, the generic application algorithm can be used to derive a type for such a call as in rule (e/tr-genapp-gabs). We show some of the resulting qualified types if *fsize* is applied to type arguments of different shapes in Figure 12.5. The type argument A is supposed to contain no dependency variables in all the examples.

The function *fsize* can be used in the same way as any other type-indexed function – it can occur on the right hand side of definitions of other type-indexed functions, and it can be used with dependency variables in the type argument to create dependencies that can be locally redefined, and it is even possible to use short notation with a function such as *fsize*.

$$\begin{array}{ll}
\mathit{fsize}\ \langle A :: * \to * \rangle & :: \forall a :: *.\quad A\ a \to \mathsf{Int} \\
\mathit{fsize}\ \langle A\ (\beta :: *) :: * \to * \rangle & :: \forall (a :: *)\ (b :: *). \\
& \quad (\mathit{size}\ \langle \beta \rangle :: b \to \mathsf{Int}) \\
& \qquad \Rightarrow A\ b\ a \to \mathsf{Int} \\
\mathit{fsize}\ \langle A\ (\beta :: *)\ (\gamma :: *) :: * \to * \rangle & :: \forall (a :: *)\ (b :: *)\ (c :: *). \\
& \quad (\mathit{size}\ \langle \beta \rangle :: b \to \mathsf{Int}, \\
& \quad\ \mathit{size}\ \langle \gamma \rangle :: c \to \mathsf{Int}) \\
& \qquad \Rightarrow A\ b\ c\ a \to \mathsf{Int} \\
\mathit{fsize}\ \langle A\ (\beta :: * \to *) :: * \to * \rangle & :: \forall (a :: *)\ (b :: * \to *). \\
& \quad (\mathit{size}\ \langle \beta\ \gamma \rangle :: \forall c :: *. \\
& \qquad\qquad (\mathit{size}\ \langle \gamma \rangle :: c \to \mathsf{Int}) \\
& \qquad\qquad\quad \Rightarrow b\ c \to \mathsf{Int}) \\
& \qquad \Rightarrow A\ b\ a \to \mathsf{Int}
\end{array}$$

Figure 12.5: Types for generic applications of *fsize* to type arguments of different form

Here are two other examples for translations of generic abstractions. The *conj* function

$$\begin{array}{ll}
\mathit{conj}\ \langle a :: * \rangle & :: (\mathit{reduce}\ \langle a \mid \mathsf{Bool} \rangle) \Rightarrow a \to \mathsf{Bool} \\
\mathit{conj}\ \langle \alpha :: * \rangle & = \mathit{reduce}\ \langle \alpha \rangle \quad \mathit{True}\ (\wedge)
\end{array}$$

translates to

$$\mathsf{cp}(\mathit{conj}, \mathsf{Abs})\ \mathsf{cp}(\mathit{reduce}, \alpha) = \mathsf{cp}(\mathit{reduce}, \alpha)\ \mathit{True}\ (\wedge)\ ,$$

and the *cata* function

$$\begin{array}{ll}
\mathit{cata}\ \langle f :: * \to * \mid a :: * \rangle & :: (\mathit{map}\ \langle f, f \rangle) \Rightarrow (f\ a \to a) \to \mathsf{Fix}\, f \to a \\
\mathit{cata}\ \langle \gamma :: * \to * \rangle\ \mathit{alg} & = \mathit{alg} \cdot \mathit{map}\ \langle \gamma \rangle \quad (\mathit{cata}\ \langle \gamma \rangle \qquad \mathit{alg}) \cdot \mathit{out}
\end{array}$$

becomes

$$\begin{array}{l}
\mathsf{cp}(\mathit{cata}, \mathsf{Abs})\ \mathsf{cp}(\mathit{map}, \gamma)\ \mathit{alg} \\
\qquad = \mathit{alg} \cdot \mathsf{cp}(\mathit{map}, \gamma)\ (\mathsf{cp}(\mathit{cata}, \mathsf{Abs})\ \mathsf{cp}(\mathit{map}, \gamma)\ \mathit{alg}) \cdot \mathit{out}
\end{array}$$

Note how the recursive call to *cata* $\langle \gamma \rangle$ can be translated without *cata* being reflexive, i.e., without *cata* having a dependency on itself.

12.5 Type indices of higher kind

Generic abstraction allows us to define type-indexed functions that are indexed by a type argument that is not of kind $*$. In the literature on generic functions, there are many examples of type-indexed functions that are defined via a type-case, but indexed by a type argument of higher kind. For example, in Hinze's POPL paper (2000*b*), one finds *map* defined as follows:

$$
\begin{array}{llll}
\mathit{map}\ \langle f :: * \to * \rangle & & :: & \forall c_1\ c_2.(c_1 \to c_2) \to f\ c_1 \to f\ c_2 \\
\mathit{map}\ \langle \mathrm{K\ Char} \rangle & f\ x & = & x \\
\mathit{map}\ \langle \mathrm{K\ Int} \rangle & f\ x & = & x \\
\mathit{map}\ \langle \mathrm{K\ Unit} \rangle & f\ x & = & x \\
\mathit{map}\ \langle \mathrm{Id} \rangle & f\ (\mathit{Id}\ x) & = & \mathit{Id}\ (f\ x) \\
\mathit{map}\ \langle \mathrm{LSum}\ \alpha\ \beta \rangle & f\ (\mathit{LInl}\ x) & = & \mathit{map}\ \langle \alpha \rangle\ f\ x \\
\mathit{map}\ \langle \mathrm{LSum}\ \alpha\ \beta \rangle & f\ (\mathit{LInr}\ x) & = & \mathit{map}\ \langle \beta \rangle\ f\ x \\
\mathit{map}\ \langle \mathrm{LProd}\ \alpha\ \beta \rangle & f\ (x_1 \otimes x_2) & = & \mathit{map}\ \langle \alpha \rangle\ f\ x_1 \otimes \mathit{map}\ \langle \beta \rangle\ f\ x_2
\end{array}
$$

Note that we use a completely different set of type constructors to define the function, all of them of kind $* \to *$, and basically lifted versions of the usual representation types:

$$
\begin{array}{ll}
\mathbf{data}\ \mathrm{K}\ (a :: *)\ (b :: *) & = K\ a \\
\mathbf{data}\ \mathrm{Id}\ (a :: *) & = \mathit{Id}\ a \\
\mathbf{data}\ \mathrm{LSum}\ (a :: * \to *)\ (b :: * \to *)\ (c :: *) & = \mathit{LInl}\ (a\ c) \mid \mathit{LInr}\ (b\ c) \\
\mathbf{data}\ \mathrm{LProd}\ (a :: * \to *)\ (b :: * \to *)\ (c :: *) & = a\ c \otimes b\ c
\end{array}
$$

We use $\otimes$ as an infix data constructor, the lifted version of $\times$. With these definitions in place, K t can be used to lift any type of kind $*$, among them Unit. The types LSum and LProd are lifted versions of Sum and Prod, respectively. Only Id is really new.

In Hinze's setting, *map*, when defined in this form, is limited to kind $* \to *$. No use of type arguments of other kind or local redefinition is possible.

It is straightforward to translate the above definition – and, in fact, any definition on such lifted type constructors – into a kind $*$ definition, augmented by a generic abstraction:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathit{map'}\ \langle a_1 :: *, a_2 :: * \mid c_1 :: *, c_2 :: * \rangle :: (\mathit{map'}\ \langle a_1, a_2 \mid c_1, c_2 \rangle)} \\
\multicolumn{4}{r}{\Rightarrow (c_1 \to c_2) \to a_1 \to a_2} \\
\mathit{map'}\ \langle \mathrm{Char} \rangle & f\ x & = & x \\
\mathit{map'}\ \langle \mathrm{Int} \rangle & f\ x & = & x \\
\mathit{map'}\ \langle \mathrm{Unit} \rangle & f\ x & = & x \\
\mathit{map'}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & f\ (\mathit{Inl}\ x) & = & \mathit{map'}\ \langle \alpha \rangle\ f\ x \\
\mathit{map'}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & f\ (\mathit{Inr}\ x) & = & \mathit{map'}\ \langle \beta \rangle\ f\ x \\
\mathit{map'}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle & f\ (x_1 \times x_2) & = & \mathit{map'}\ \langle \alpha \rangle\ f\ x_1 \times \mathit{map'}\ \langle \beta \rangle\ f\ x_2 \\
\mathit{mapId} & f\ x & = & f\ x
\end{array}
$$

$$\begin{array}{lll}
\mathit{map}\ \langle f :: * \to * \mid c_1 :: *, c_2 :: * \rangle & :: & (\mathit{map}'\ \langle f,f \mid c_1, c_2 \rangle) \\
 & & \quad \Rightarrow (c_1 \to c_2) \to f\ c_1 \to f\ c_2 \\
\mathit{map}\ \langle \gamma :: * \to * \rangle\ f\ x & = & \mathbf{let}\ \mathit{map}'\ \langle \alpha \rangle = \mathit{mapId} \\
 & & \mathbf{in}\ \ \mathit{map}'\ \langle \gamma\ \alpha \rangle\ f\ x
\end{array}$$

For each case but Id, we have a direct corresponding case for the unlifted version of the type. The identity case can be defined as a separate function, here *mapId*, that is subsequently plugged into the locally redefined element position α of the type argument γ in *map*, that is defined via generic abstraction.

The rewritten version of *map* has the advantage that we can apply short notation to apply it to type arguments of arbitrary kind, or, more generally, that we can use local redefinition to modify its behaviour. This version of *map* differs from the *map* that we have defined in Section 9.1 only in the presence of the function argument f that is passed everywhere. A closer look reveals that it is indeed not needed anywhere but in *mapId*, and thus can be optimized away.

12.6 Multiple type arguments

In the general case, type-indexed functions with multiple type arguments are not a trivial extension. The reason is that for multiple type arguments, we get an explosion of dependencies: for one type argument, we need one dependency constraint per function we depend on and dependency variable. With two type arguments, we already have one per function and pair of dependency variables. For instance, *poly* $\langle \mathrm{Sum}\ \alpha_1\ \beta_1 \rangle\ \langle \mathrm{Sum}\ \alpha_2\ \beta_2 \rangle$ could – because four dependency variables occur – depend up to $4^2 \equiv 16$ times on *poly*.

Allowing different dependencies for each case in a type-indexed function definition could help, but would be a significant change of the theory and make the type system yet more complex. The exact implications of such an approach are future work.

Perhaps surprisingly, though, multiple type arguments are easy to allow for functions defined using generic abstraction. The reason is that generic abstractions do not occur in dependency constraints, hence there is no combinatorial explosion of dependencies.

The following function defines a generic variant of the standard list function *elem* that determines whether an element is contained in a data structure.

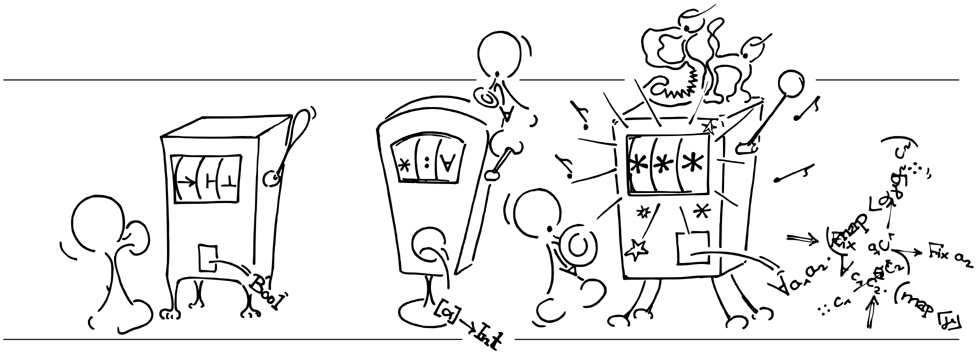
$$\begin{array}{lll}
\mathit{elem}\ \langle a :: * \rangle\ \langle f :: * \to * \rangle & :: & (\mathit{enum}\ \langle a \rangle, \mathit{equal}\ \langle a \rangle, \mathit{reduce}\ \langle f \mid \mathrm{Bool} \rangle) \\
 & & \quad \Rightarrow a \to f\ a \to \mathrm{Bool} \\
\mathit{elem}\ \langle \alpha :: * \rangle\ \langle \gamma :: * \to * \rangle\ x & = & \mathit{any}\ \langle \gamma \rangle\ (\mathit{equal}\ \langle \alpha \rangle\ x)
\end{array}$$

We must be able to apply *reduce* to the data structure, because we use *any* in the implementation of the function, and *any* is defined via generic abstraction over *reduce*. The function *any* takes a predicate and checks if the predicate holds for at least one element in the structure. As predicate, we take the partially applied *equal* function.

The translation is simple to adapt. For *elem*, we get

$$\begin{aligned}&\mathsf{cp}(\mathit{elem},\mathsf{Abs})\ \mathsf{cp}(\mathit{enum},\alpha)\ \mathsf{cp}(\mathit{equal},\alpha)\ \mathsf{cp}(\mathit{reduce},\gamma)\ x = \\ &\qquad\qquad\qquad\mathsf{cp}(\mathit{any},\mathsf{Abs})\ \mathsf{cp}(\mathit{reduce},\gamma)\ (\mathsf{cp}(\mathit{equal},\alpha)\ x)\ .\end{aligned}$$

13 Type Inference



So far in this thesis, we have completely ignored type inference. We have assumed completely type-annotated programs and have only presented rules that can check programs to be type correct. For practical programming, this is, however, inadequate. One of the great strengths of Haskell is that, as it is based on the Hindley-Milner type system (Hindley 1969; Milner 1978), type annotations are rarely required.

Even in standard Haskell we sometimes need type signatures: for polymorphic recursion, to circumvent the dreaded monomorphism restriction, or for the methods of type classes, we have to explicitly specify a signature to give the type checker a nudge into the right direction. If arbitrary-rank polymorphic types (Peyton Jones and Shields 2003) are allowed, then functions involving such types need explicit annotations as well.

Therefore, one option to tackle the type inference of type-indexed functions is simply not to do it. Type-indexed functions have mandatory type signatures, and so do functions defined via generic abstraction. At the call sites of type-indexed functions, we always pass an explicit type argument anyway. At all other places, we can use ordinary Haskell type inference and do not have to annotate. This is a simple and pragmatic solution that works. Generic functions are defined once and used often, thus the overhead of giving a type signature during definition seems bearable.

On the other hand, the presence of dependencies makes type signatures of type-indexed functions nontrivial, because the theory of dependencies must be known to the programmer. Furthermore, the requirement that type arguments must be passed at the call sites of generic functions is annoying. It is, for example, much more pleasant to use a class method, where the compiler usually infers the instances to use.

In this chapter, we distinguish and formulate a number of different inference problems in the context of Generic Haskell, and relate them to other problems or sketch how they might be solved.

In Section 13.1, we will discuss inference of type arguments of type-indexed functions, so that we can, for example, write *equal* 4 2 instead of *equal* ⟨Int⟩ 4 2. In Section 13.2, we will discuss how dependencies of type-indexed functions could be inferred automatically. This would simplify the type signatures that the programmer has to provide and make it easier to start with generic programming because one does not have to delve into the theory of dependencies before doing so. Last, in Section 13.3 we look into possibilities for inferring the base type of a type-indexed function. If both dependency inference and inference of the base type were possible, no explicit type signatures for type-indexed functions would be needed at all. However, base type inference turns out to be difficult, because in general, there is no such thing as a *principal type* for a type-indexed function.

13.1 Type inference of type arguments

In all languages that we have introduced, generic application – the application of a type-indexed function to a type argument – is explicit: a type argument has to be passed to specify to which type the function in question is to be instantiated.

This seems adequate in cases where dependency variables occur in the type argument – after all, we might want to locally redefine the function for some of these dependency variables, so it is better to be precise about the details. Recall the example

```
let size ⟨α⟩ = const 1
in  (size ⟨[[Int]]⟩ [[1,2,3],[4,5]],
     size ⟨[[α]]⟩   [[1,2,3],[4,5]],
     size ⟨[α]⟩     [[1,2,3],[4,5]],
     size ⟨α⟩       [[1,2,3],[4,5]])
```

from Section 8.2, where only the type arguments distinguish the four calls, indicating which part of the data structure we want to count.

However, most calls of type-indexed functions in a program probably have constant (i.e., dependency variable free) type arguments, and for constant types, it is desirable to be allowed to omit the type argument.

If we use a class method of a Haskell type argument, the compiler infers the dictionary to use, i.e., the combination of type class instances that provide the desired functionality. We write *show* `'c'` or *show False* or *show* [1] without having to specify explicitly on which type *show* should be invoked. The compiler detects automatically, using context information, that we intend to call *show* on Char, Bool, and [Int], respectively. For type classes, there is no way to specify this implicit argument explicitly, thus if the inference fails, there is no way to pass the missing information to the type checker.

The type inference problem for type arguments of type-indexed functions is very similar to the one for type classes in Haskell. Provided that all generic type parameters of a type-indexed function's type signature actually occur in the base type of the function (functions like *enum* or *card* do not fulfill this condition), we can apply the same technique that the Haskell compiler uses to infer the correct dictionary for a type class method.

An important difference is, though, that even in the presence of type inference for type arguments, we still allow to explicitly specify a type argument, be it for documentation purposes, or because we want to make use of dependency variables or local redefinitions, or to bypass restrictions (such as for *enum* or *card*).

The type class encoding, presented as an alternative translation for generic functions in Section 11.8.2, demonstrates the relation between the inference of type arguments and the inference of dictionaries for type classes further.

13.2 Dependency inference

Dependency inference amounts to partially inferring the type signatures of type-indexed functions. We still require the base type of a generic function to be specified, but we would like to infer the dependency list automatically.

For example, the type signature of *equal* could then be written as

$$\mathit{equal}\ \langle a :: *\rangle :: a \to a \to \text{Bool}\ ,$$

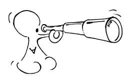
and the one for *map* would become

$$\mathit{map}\ \langle a :: *, b :: *\rangle :: a \to b\ \ .$$

This would yet again significantly reduce the amount of knowledge the programmer needs to have about the underlying type system.

The problem is easy to solve in the situation of Chapter 6, where we first introduced dependencies. There, we have treated the special case that types of type-indexed functions are always of arity $\langle 1 \mid 0 \rangle$, i.e., they have one generic, and no non-generic arguments. In this situation, there is only one way in which a type-indexed function can depend on another. Therefore, it suffices to infer the *names* of the functions that are dependencies. We can achieve this by scanning the definitions of type-indexed functions and determining the minimal set of dependencies for each functions that is sufficient.

Although we believe that dependency inference is possible also for the general case of Chapter 9 – at least partially – this remains future work.



13.3 Base type inference

In the presence of polymorphism and subtyping, an expression often can have more than one type. A function such as

$$\lambda x\ y \to x$$

can be assigned any of the types

$$\begin{array}{llll} & \mathrm{Int} & \to \mathrm{Char} & \to \mathrm{Int} \\ & \mathrm{Bool} & \to \mathrm{Bool} & \to \mathrm{Bool} \\ \forall a :: *. & \mathrm{Tree}\ a & \to [a] & \to \mathrm{Tree}\ a \\ \forall (a :: *)\ (b :: *). & a & \to b & \to a\ . \end{array}$$

The type system on which Haskell is based (Damas and Milner 1982) has the advantage that there always exists a most general type, the **principal type**, which is convertible into all other types that the expression can have. Formally, if an expression e has principal type t, then if also $e :: t'$, then $t \leqslant t'$, where $t \leqslant t'$ means that every t expression can be used as a t' expression. In the example of $\lambda x\ y \to x$, the last of the four types given above is the principal type.

The type inference system for Haskell is only really helpful because it finds and returns the principal type for every expression. This way, we can run the inference algorithm on single functions to learn about their types, and we can be sure that the types that are inferred are the best we can come up with.

In this section, we discuss the question whether there exist principal types for generic functions. Unfortunately, the answer is no, and we will demonstrate this negative answer with an example.

Recall the introduction of function *map* in Section 9.1. We obtained *map* as a generalization of the generic identity function *gid*, by using a more general type signature, but without touching the implementation. The type signatures of *gid* and *map* are

$$\begin{array}{l} gid\ \langle a \qquad :: *\rangle :: (gid\ \ \langle a\rangle) \qquad \Rightarrow a \ \ \to a \\ map\ \langle a_1, a_2 :: *\rangle :: (map\ \langle a_1, a_2\rangle) \Rightarrow a_1 \to a_2\ . \end{array}$$

Both functions have exactly the same code, i.e., the type of *gid* would also work for *map* (renaming the dependency, of course), and vice versa. Using local redefinition, both can be used to map a function over a data structure, but with *map* we can change the type of the elements, whereas with *gid* we cannot.

Let us analyze which of the two types a type inferencer should return for a function that has the implementation of *gid* or *map*. Certainly, the second type seems "better", somehow, as in practice, *map* is far more useful than *gid* is.

However, we will demonstrate now that neither of the two types is more general. The type of *gid* cannot be used for *map* in the following situation:

$$map\ \langle[\,]\rangle\ chr\ [1,2,3,4,5]\ .$$

This is a simple application on lists that changes the element type.

The other direction requires a datatype that takes a type argument of higher kind to be involved, a datatype such as GRose:

$$\textbf{data}\ \text{GRose}\ (f :: * \to *)\ (a :: *) = GFork\ a\ (f\ a)\ .$$

This is a generalization of the datatype Rose of rose trees, defined on page 158 in Section 10.2. Here, the children can be stored in an arbitrary data structure f, whereas in a rose tree, $f \equiv [\,]$. Now, consider the expression

$$\begin{array}{l} \textbf{let}\ gid\ \langle\gamma\ (\alpha :: *)\rangle\ z = \textbf{case}\ z\ \textbf{of} \\ \qquad\qquad\qquad\qquad\qquad\quad [\,] \qquad \to [\,] \\ \qquad\qquad\qquad\qquad\qquad\quad (x : y) \to gid\ \langle\alpha\rangle\ (gid\ \langle\alpha\rangle\ x) : gid\ \langle\gamma\ \alpha\rangle\ y \\ \textbf{in}\ \ gid\ \langle\text{GRose}\ \gamma\ \text{Int}\rangle\ . \end{array}$$

The redefinition uses γ at the list type constructor. In the second arm of the case construct, the funcion *gid* $\langle\alpha\rangle$ is applied twice to the head of the list. We therefore rely on the fact that *gid* $\langle\alpha\rangle$ in the local redefinition does *not* change the type of its argument. We could not make this assumption if *gid* had the type of *map*; the redefinition *gid* $\langle\gamma\ \alpha\rangle$ would then need to be of the type

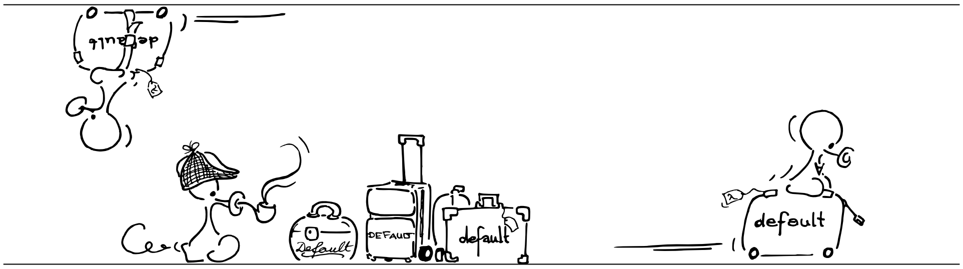
$$\begin{array}{l} \forall (c_1 :: * \to *)\ (c_2 :: * \to *)\ (a_1 :: *)\ (a_2 :: *). \\ \quad (gid\ \langle\alpha\rangle :: a \to b) \Rightarrow c_1\ a_1 \to c_2\ a_2\ . \end{array}$$

Compare this situation with the four types for $\lambda x\ y \rightarrow x$ that we have given above. We could always replace one of the first three types for the function with the fourth, without breaking programs that use the function.

The whole consideration has an effect for type inference of the type signature of a type-indexed function: we cannot hope to find a best possible type in general. If a function with the definition such as *map* and *gid* is encountered, it is unclear whether to assign it *gid*'s or *map*'s type. We could still try to find a type (or, extend the type system and permit a type) which is convertible into any of the two other types, but it is not obvious what this type would look like.

Even in the knowledge of this negative result, we do not have to give up completely, though. For example, the above problem does only occur if we consider functions with more than one generic type variable in the type tuple, i.e., functions of arity $\langle r \mid s \rangle$ where $r > 1$. If we restrict ourselves to the case $r \equiv 1$, and possibly impose further restrictions, we believe that type inference can be done. Even for the case that $r > 1$, it might be possible to infer a type under certain circumstances, or to develop a type-inference algorithm that returns more than one type. We might even choose to prefer the type of *map* over the type of *gid* anyway, because examples where the more restrictive type can prove useful require local redefinition on types of higher kinds, a situation which is not likely to occur very frequently. All this is future work.

14 Default Cases



Experience with generic programming shows that one often writes several minor variations of one generic function over and over again. The reason is that while one is interested in generic behaviour for the large part of the datatypes involved, some particular types should be treated in a special way. An example is the collection of variables from an abstract syntax tree, where the algorithm is basically the generic function *collect* (defined on page 137 in Section 9.3), but occurrences of the datatype Var that holds variables should be added to the list that is collected.

A first approach to solve the problem would probably be to locally redefine *collect* somehow and capture the redefined functionality in a generic abstraction:

$$\begin{aligned} &\mathit{varcollect}\ \langle \alpha :: * \rangle = \mathbf{let}\ \mathit{collect}\ \langle \alpha \rangle\ (V\ x) = [x] \\ &\qquad\qquad\qquad\qquad\ \mathbf{in}\ \ \mathit{collect}\ \langle \mathrm{Expr}\ \alpha \rangle\ , \end{aligned}$$

assuming that Expr is the datatype that holds an expression of the language in question. This definition is not ideal, because it requires Expr to be parametrized over the type of variables. If we subsequently want to collect something else, say type signatures, then we have to parametrize over those as well. It is not acceptable to change the datatype just to be able to write another function, nor can we necessarily predict which functions might be needed while designing the datatype.

In this chapter, we introduce **default cases**, which provide another solution to this dilemma: we can extend an already existing type-indexed function by adding new cases or even overriding existing ones. For the collection of variables, we can then write

$$\begin{array}{l}
\mathit{varcollect}\ \textbf{extends}\ \mathit{collect} \\
\mathit{varcollect}\ \langle \text{Var} \rangle\ (V\ x) = [x]
\end{array}$$

to achieve the desired effect.

In the following section, we build upon the variable collection example and discuss some other examples in which default cases can be applied successfully. After that, we take a look at the corresponding language extension and its semantics.

14.1 Generic Traversals

Let us assume the following system of datatypes which describe a simple expression language based on the lambda calculus.

$$\begin{array}{lcll}
\textbf{data}\ \text{Var} & = & V & \text{String} \\
\textbf{data}\ \text{Type} & = & \mathit{TyVar} & \text{Var} \\
 & \mid & \mathit{Fun} & \text{Type Type} \\
\textbf{data}\ \text{Expr} & = & \mathit{Var} & \text{Var} \\
 & \mid & \mathit{App} & \text{Expr Expr} \\
 & \mid & \mathit{Lam} & (\text{Var}, \text{Type})\ \text{Expr} \\
 & \mid & \mathit{Let} & (\text{Var}, \text{Type})\ \text{Expr Expr}
\end{array}$$

Using a default case, a function to collect all variables from an expression can be written as a type-indexed function:

$$\begin{array}{l}
\mathit{varcollect}\ \langle a :: * \rangle :: (\mathit{varcollect}\ \langle a \rangle) \Rightarrow a \to [\text{Var}] \\
\mathit{varcollect}\ \textbf{extends}\ \mathit{collect} \\
\mathit{varcollect}\ \langle \text{Var} \rangle \qquad x \qquad\quad = x \\
\mathit{varcollect}\ \langle \text{Prod}\ \alpha\ \beta \rangle\ (x_1 \times x_2) = \mathit{varcollect}\ \langle \alpha \rangle\ x_1\ \text{`}\mathit{union}\text{`}\ \mathit{varcollect}\ \langle \beta \rangle\ x_2
\end{array}$$

In comparison with the example from the introduction to this chapter, we give a full definition here, including a type signature, and we add yet another case, *redefining* the behaviour for products to use set union and eliminate duplicates instead of list concatenation. Note that the function does extend *collect*, but does not depend on it.

Even though this function is intended to be used on the type Expr, it is still generic. It is robust against changes in the definitions of datatypes, it will work for other languages that make use of the Var type, and it will work for data structures that contain expressions, such as [Expr] or Tree Expr, all for free.

The following definition of *varcollect* is semantically equivalent to the one above and shows how *varcollect* would have to be written in absence of the default case construct:

$$\begin{array}{lll}
\mathit{varcollect}\ \langle a :: *\rangle :: (\mathit{varcollect}\ \langle a\rangle) \Rightarrow a \rightarrow [\mathrm{String}] & & \\
\mathit{varcollect}\ \langle \mathrm{Var}\rangle\quad (V\ x) & = & [x] \\
\mathit{varcollect}\ \langle \mathrm{Prod}\ \alpha\ \beta\rangle\ (x_1 \times x_2) & = & \mathit{varcollect}\ \langle\alpha\rangle\ x_1\ \text{`}\mathit{union}\text{'}\ \mathit{varcollect}\ \langle\beta\rangle\ x_2 \\
\mathit{varcollect}\ \langle \mathrm{Int}\rangle & = & \mathit{collect}\ \langle \mathrm{Int}\rangle \\
\mathit{varcollect}\ \langle \mathrm{Unit}\rangle & = & \mathit{collect}\ \langle \mathrm{Unit}\rangle \\
\mathit{varcollect}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle & = & \mathbf{let}\ \mathit{collect}\ \langle\alpha\rangle = \mathit{varcollect}\ \langle\alpha\rangle \\
 & & \qquad \mathit{collect}\ \langle\beta\rangle = \mathit{varcollect}\ \langle\beta\rangle \\
 & & \mathbf{in}\ \ \mathit{collect}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle
\end{array}$$

From this equivalent definition, we can infer how the default case works: for each type that is in the signature of *collect*, but for which there is no explicit arm in *varcollect*, the compiler maps the *varcollect* call to a call of *collect*. Most interesting is the case for Sum: the compiler automatically remaps the dependencies that *collect* has on itself to point to *varcollect*. This way, the extension is *deep*: all occurrences of the Var type will be collected, and there is no dependency on the original *collect* function.

Type patterns of type-indexed functions always consist of a named type applied to dependency variables (cf. Figure 6.1). The order of cases in a **typecase** is unimportant, because we can always determine the matching case by analyzing the toplevel constructor of the type argument. For default cases, it is therefore unimportant whether the new cases are inserted in the beginning, the end, or somewhere in the middle of the **typecase** construct – it makes no difference.

It is also possible to reuse arms from an existing definition in a new function, even arms that are overridden. The following function collects just term variables from our expression, assuming a function *termvar* :: Var → Bool to determine whether a variable comes from the syntactic category of term variables:

$$\begin{array}{l}
\mathit{termcollect}\ \langle a :: *\rangle :: (\mathit{termcollect}\ \langle a\rangle) \Rightarrow a \rightarrow [\mathrm{Var}] \\
\mathit{termcollect}\ \mathbf{extends}\ \mathit{varcollect} \\
\mathit{termcollect}\ \langle \mathrm{Var}\rangle\ v = \mathbf{if}\ \mathit{termvar}\ v\ \mathbf{then}\ \mathit{varcollect}\ \langle \mathrm{Var}\rangle\ v\ \mathbf{else}\ [\,]\ .
\end{array}$$

Based on *termcollect*, we can then write a function that determines the free variables in an expression:

```
$freecollect\ \langle a :: *\rangle :: (freecollect\ \langle a\rangle) \Rightarrow a \to [\mathrm{Var}]$
$freecollect$ extends $termcollect$
$freecollect\ \langle \mathrm{Expr}\rangle\ e =$
  case $e$ of
    $Lam\ (v,t)\ e'$      $\to filter\ (\not\equiv v)\ (freecollect\ \langle \mathrm{Expr}\rangle\ e')$
    $Let\ \ (v,t)\ e'\ e''$ $\to freecollect\ \langle \mathrm{Expr}\rangle\ e'$
                          $+\!\!+\ filter\ \langle \not\equiv v\rangle\ (freecollect\ \langle \mathrm{Expr}\rangle\ e'')$
    $\_$                  $\to termcollect\ \langle \mathrm{Expr}\rangle\ e$
```

In general, operations on larger systems of datatypes such as abstract syntax trees, datatypes representing XML DTDs or schemata (W3C 2001), or datatypes representing an organizational structure, such as a company, a package database, or a file system, are good candidates for processing with generic functions defined by means of default cases. A library of basic traversals such as *collect* or *map* can be used to define several variations that perform the tasks at hand and adapt to changes in the data structures automatically.

Ralf Lämmel has studied such generic traversals in detail (Lämmel *et al.* 2000), also in the context of Haskell (Lämmel and Visser 2002). This line of research has recently culminated in a convincing proposal to extend the Haskell language with support for generic traversals and queries (Lämmel and Peyton Jones 2003). Compared with Generic Haskell, this language extension puts the focus on traversal functions and the unproblematic extension of existing functions with specific behaviour, and puts less emphasis on the generic structure of types. In a more recent version, it is now also possible to define a larger class of classic generic operations such as equality or show in this setting (Lämmel and Peyton Jones 2004).

14.2 Variants of equality

Another example where default cases come in handy is to define a variant of an existing standard function, such as equality. In Section 7.3, we already discussed that we have the possibility to define special behaviour for a datatype, and as an example we added the arm

$$equal\ \langle \mathrm{Range}\rangle\ x\ y = True\ .$$

However, this definition robs us of the possibility to use the generic, structural equality, on the Range type. Maybe we are interested in ignoring ranges in data structures sometimes, but not always. Even worse, we may not yet know about the Range type and our specific application when defining the equality function – especially if the program consists of multiple modules (cf. Chapter 18).

An alternative is to create a new function that extends *equal*:

$$\begin{aligned}&\mathit{requal}\ \langle a :: *\rangle :: (\mathit{enum}\ \langle a\rangle, \mathit{requal}\ \langle a\rangle) \Rightarrow a \to a \to \mathrm{Bool}\\ &\mathit{requal}\ \textbf{extends}\ \mathit{equal}\\ &\mathit{requal}\ \langle \mathrm{Range}\rangle\ x\ y = \mathit{True}\ \ .\end{aligned}$$

Another variant of *equal* results from comparing characters case-insensitively:

$$\begin{aligned}&\mathit{ciequal}\ \langle a :: *\rangle :: (\mathit{enum}\ \langle a\rangle, \mathit{requal}\ \langle a\rangle) \Rightarrow a \to a \to \mathrm{Bool}\\ &\mathit{ciequal}\ \textbf{extends}\ \mathit{equal}\\ &\mathit{ciequal}\ \langle \mathrm{Char}\rangle\ x\ y = \mathit{toUpper}\ x \equiv \mathit{toUpper}\ y\ \ .\end{aligned}$$

Several other variations of *equal* are possible and may be useful occasionally.

14.3 Simulating multiple dependencies on one function

In Section 9.8, we promised that default cases would provide a solution for type-indexed functions that depend on the same function more than once.

As an example, let us use the *map* function. In Section 11.2, we have introduced the *bimap* function that produces a pair of functions EP a b instead of a single function $a \to b$, mainly because we could then easily lift *bimap* to function types.

An alternative, sketched in Section 11.3, is to define two copies of *map*, there called *from* and *to*, which depend on each other in the function case.

Had we multiple dependencies on one function, we could define *map* for functions directly:

$$\mathit{map}\ \langle a :: *, b :: *\rangle :: (\mathit{map}\ \langle a, b\rangle, \mathit{map}\ \langle b, a\rangle\ \textbf{as}\ \mathit{comap}) \Rightarrow a \to b\ \ .$$

We depend on *map* as usual, and once more with reversed type variables and using the name *comap*. The definition would be as usual, plus the case for the function type constructor, namely

$$\mathit{map}\ \langle \alpha \to \beta\rangle\ x = \mathit{map}\ \langle \beta\rangle \cdot x \cdot \mathit{comap}\ \langle \alpha\rangle$$

In the function case, we need an arrow in the other direction, corresponding to the contravariance of the first argument of the function arrow.

Using default cases, we can achieve the same goal without an additional renaming construct in the syntax of type signatures. We adapt the type signature of *map* to refer to *comap* as a different function:

$$map \quad \langle a :: *, b :: * \rangle :: (map\ \langle a, b \rangle, comap\ \langle b, a \rangle) \Rightarrow a \to b \ .$$

The definition of *map* is exactly as before. We use a default case to define *comap* as a copy of *map*:

```
comap ⟨b :: ∗, a :: ∗⟩ :: (map ⟨a, b⟩, comap ⟨b, a⟩) ⇒ a → b
comap extends map
  where map   as comap
        comap as map .
```

Note that we swap the dependencies to *map* and *comap* in the definition of *comap*, using a **where** clause to the **extends** construct.

The compiler now treats *comap* $\langle \alpha \to \beta \rangle$ as if it were defined as

```
comap ⟨α → β⟩ = letsim map   ⟨α⟩ = comap ⟨α⟩
                       map   ⟨β⟩ = comap ⟨β⟩
                       comap ⟨α⟩ = map   ⟨α⟩
                       comap ⟨β⟩ = map   ⟨β⟩
                in     map ⟨α → β⟩ ,
```

where the **letsim** construct is a simultaneous, non-recursive **let**: all assignments are performed at the same time, and the right hand sides can only refer to objects on the outside of the **let**. The compiler can translate a **letsim** into an ordinary, nested **let** statement, making use of temporary variables, or simply by performing a simultaneous substitution. The definition above is thus not mutually recursive. The component belonging to the function case of *bimap* is:

$$\begin{aligned}&\mathsf{cp}(comap, (\to))\ \mathsf{cp}(comap, \alpha)\ \mathsf{cp}(map, \alpha)\ \mathsf{cp}(comap, \beta)\ \mathsf{cp}(map, \beta) = \\ &\quad \mathsf{cp}(map, (\to))\ \mathsf{cp}(map, \alpha)\ \mathsf{cp}(comap, \alpha)\ \mathsf{cp}(map, \beta)\ \mathsf{cp}(comap, \beta) \ .\end{aligned}$$

A **where** clause in the default case such as the one in the definition of *comap* can generally be used in a default case to rename the dependencies of the extended function.

A normal **extends** clause without a where-part, as discussed in all the previous examples, is a special case, where we assume that we want to rename a reflexive dependency to point to the new function. Thus, the statement

```
x extends y
```

is short for

$$\begin{array}{l} x \ \textbf{extends}\ y \\ \quad \textbf{where}\ y\ \textbf{as}\ x \ . \end{array}$$

If the function being extended has no dependency on itself, then there is no implicit renaming of dependencies.

14.4 Implementation of default cases

Value declarations

$$\begin{array}{lll} d & ::= \ldots & \text{everything from Figures 4.1, 8.3, and 12.1} \\ & \mid\ x' \ \textbf{extends}\ x\ \textbf{where}\ \{y_k\ \textbf{as}\ y'_k\}_{;}^{k \in 1..\ell} & \\ & & \text{default case} \end{array}$$

Figure 14.1: Core language with default cases FCR+gf+gabs+dc, extends language FCR+gf+gabs in Figures 12.1, 9.4, 8.3, 6.1, 4.1, and 3.1

To implement default cases, we need only a very simple extension of the syntax, that is shown in Figure 14.1. We introduce a new form of value declarations, that augments a type-indexed function definition in a **typecase**. A default case thus only makes if it is paired with a (possibly empty) **typecase**.

Default cases are translated to a number of components for the type-indexed function that they refer to. To define precisely which components are generated for a default case, we have to clarify the notion of a type-indexed function's *signature* a bit: the **presignature** of a function is the set of type constructors that the programmer has written arms for in the typecase of the function definition. The **signature** consists of the type constructors that make up the presignature plus the type constructors for which additional components are defined via a default case. In particular, for a type-indexed function that is not defined with a default case, presignature and signature refer to the same set of type constructors.

If x' **extends** x, then the signature of x' is the union of the presignature of x' and the signature of x. As a formula, this is

$$x' \ \textbf{extends}\ x \Longrightarrow \mathsf{signature}(x') \equiv \mathsf{signature}(x), \mathsf{presignature}(x') \ .$$

The cases that make up the presignature of a function x' are always taken from x'. The type constructors that are in the signature of x, but not in the presignature of x', are the ones that the default case expands to.

We assume that every entry in the signature environment Σ of the form $x \ \langle T \rangle$ is marked with either pre, dc, or gf. All arms that are explicitly defined in the

$$[\![d_{\text{FCR+gf+gabs+dc}} \rightsquigarrow \Gamma_2; \Sigma_2]\!]^{\text{dc}}_{K;\Gamma_1;\Delta;\Sigma_1;\Psi} \equiv \{d_{\text{FCR}\,i}\}^{i\in 1..n}_{;}$$

$$\frac{\begin{array}{c} \mathsf{signature}_\Sigma(x) - \mathsf{presignature}_\Sigma(x') \equiv \{T_i\}^{i\in 1..n}_{,} \\ \{\mathsf{convert}_{K;\Gamma}(x, x', T_i, \mathsf{new}) \equiv x' \ \langle P_i\rangle = e_i\}^{i\in 1..n} \\ \{[\![x' \mid P_i \to e_i \rightsquigarrow x' \ \langle T_i\rangle]\!]^{\text{dc}}_{K;\Gamma;\Delta;\Sigma;\Psi} \equiv d_i\}^{i\in 1..n} \\ \mathsf{new} \equiv \{y_k \mapsto y'_k\}^{k\in 1..\ell}_{.} \\ \Sigma' \equiv \{x \ \langle T_i\rangle \ \mathsf{dc}\}^{i\in 1..n}_{,} \end{array}}{[\![x' \ \textbf{extends} \ x \ \textbf{where} \ \{y_k \ \textbf{as} \ y'_k\}^{k\in 1..\ell}_{;} \rightsquigarrow \varepsilon; \Sigma']\!]^{\text{dc}}_{K;\Gamma;\Delta;\Sigma;\Psi} \equiv \{d_i\}^{i\in 1..n}_{;}} \quad \text{(d/tr-dc)}$$

Figure 14.2: Translation of default cases, extends Figures 11.2 and 6.15

program are marked with pre and make up the presignature. All arms that are added by a default case are marked with dc. The cases marked with pre together with the cases marked with dc make up the signature of a function. Finally, all entries that stem from the generation of an additional component for a generic function are marked with gf. They are of no direct interest for the translation of default cases. Assuming that these tags are present in the signature environment, we can define functions of the form

$$\begin{aligned} \mathsf{signature}_\Sigma(x) &\equiv \{T_i\}^{i\in 1..n}_{,} \\ \mathsf{presignature}_\Sigma(x) &\equiv \{T_i\}^{i\in 1..n}_{,} \end{aligned}$$

that return the signature or presignature of a type-indexed function x by scanning the environment Σ. The tags are ignored by the generic translation algorithm, for which it is unimportant where a specific component comes from.

In Figure 14.2, we learn how a default case is formally translated. As we have explained, the **extends** construct generates arms for a certain number of type constructors. These are computed by subtracting the presignature of the new function x' from the signature of the extended function x. These type constructors are called T_i. For each of the T_i, we call the convert function to perform the renaming, including the renaming of the dependencies. The convert function takes four arguments: the old function x, the new function x', the type constructor in question T_i, and a function called new that maps names of dependencies of the original function to names of dependencies of the new function. The function new is built from the **where** clause of the **extends** construct.

The convert function returns a declaration of the form $x' \ \langle P_i\rangle = e_i$, where P_i is a type pattern built from T_i. We reformat this declaration as if it were an arm

$$\mathsf{convert}_{K;\Gamma}(x, x', A, \mathsf{new}) \equiv d$$

$$\frac{\begin{array}{c} K \vdash A :: \{\kappa_i \rightarrow\}^{i \in 1..n}\ \kappa \\ \{\alpha_i \text{ fresh}\}^{i \in 1..n} \qquad K' \equiv K\ \{, \alpha_i :: \kappa_i\}^{i \in 1..n} \\ \mathsf{dependencies}_{\Gamma}(x) \equiv \{y_k\}_{,}^{k \in 1..\ell} \end{array}}{\begin{array}{l} \mathsf{convert}_{K;\Gamma}(x, x', A, \mathsf{new}) \equiv \\ \quad x'\ \langle A\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle = \\ \qquad \textbf{letsim}\ \big\{\{\mathsf{convert}_{K';\Gamma}(\mathsf{new}(y_k), y_k, \alpha_i, \mathsf{new}^{-1})\}^{i \in 1..n}\big\}^{k \in 1..\ell} \\ \qquad \textbf{in} \qquad x\ \langle A\ \{\alpha_i\}^{i \in 1..n} \rangle\ . \end{array}} \quad \text{(dc-convert)}$$

Figure 14.3: Conversion of arms for a default case

of the typecase $P_i \rightarrow e_i$, and use the old rule (d/tr-arm) to translate the arm to a declaration d_i for a single component $\mathsf{cp}(x, T_i)$. Note that this translation also type checks the declaration d_i. We record in the signature environment that $x\ \langle T_i \rangle$ has been generated by a default case, marking the entry with dc.

The component declarations d_i resulting from the converted arms are returned together with the signature environment and an empty type environment, because no new FCR+gf+gabs+dc functions are declared by a default case.

The convert function is defined in Figure 14.3. An application looks as follows:

$$\mathsf{convert}_{K;\Gamma}(x, x', A, \mathsf{new}) \equiv d\ .$$

It takes a type argument A (which should be either a named type or a single dependency variable), an old and a new function name x and x', and a map from old dependencies to new dependencies, which we call new. It produces a declaration. The conversion takes place under environments K and Γ, because we need access to the dependencies of the functions involved.

There is only one rule, (dc-convert). It makes use of a **letsim** construct in the result, which is like a **let** construct, but performs all bindings simultaneously and non-recursively. It is easy to translate a **letsim** construct performing a simultaneous substitution of the bound variables. An example of in which situations the **letsim** construct is necessary and how it can be translated was given on page 220. We use the construct here to simplify the rule, but do not formally define it. The rule (dc-convert) checks the kind of the type argument A, and introduces dependency variables α_i as arguments for A, of appropriate kinds. We cannot, in general, write

$$x' \ \langle A \ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n}\rangle = x \ \langle A \ \{\alpha_i\}^{i \in 1..n}\rangle \ .$$

because that would ignore possible dependencies of x. The dependencies of x should be remapped to dependencies of x' according to the function new. We will perform the remapping by using local redefinition around the call $x \ \langle A \ \{\alpha_i\}^{i \in 1..n}\rangle$. We must redefine every dependency y_k of x in terms of the function new(y_k), as specified by the function new, and this for every dependency variable α_i that occurs in the type pattern. It turns out that we can use convert recursively for this job. We only need to invert the mapping new, and we write the inverse mapping new^{-1}.

Let us assume for the moment that this inverse exists and have a look at an example. We had

varcollect **extends** *collect*

which with explicit **where** clause corresponds to

varcollect **extends** *collect*
  where *collect* **as** *varcollect* .

The function new that remaps the dependencies is in this case thus *collect* $\mapsto$ *varcollect*.

For a case of kind $*$ such as Int, we get

convert$(\mathit{collect}, \mathit{varcollect}, \text{Int}, \mathit{collect} \mapsto \mathit{varcollect}) \equiv$
  $\mathit{varcollect} \ \langle \text{Int}\rangle = \mathit{collect} \ \langle \text{Int}\rangle$.

Because there are no dependency variables involved, we also do not have any dependencies to remap. We omit the empty **let** construct. For a type of the kind $*^n$ for $n > 0$, such as Sum, we can witness the remapping of the dependencies:

```
convert(collect, varcollect, Sum, collect ↦ varcollect) ≡
  varcollect ⟨Sum (α :: ∗) (β :: ∗)⟩ =
                    letsim collect ⟨α⟩ = varcollect ⟨α⟩
                           collect ⟨β⟩ = varcollect ⟨β⟩
                    in     collect ⟨Sum α β⟩ .
```

Note that the inverse mapping new^{-1}, which would be *varcollect* $\mapsto$ *collect*, is not needed in this case. The dependency variables are all of kind $*$, and thus have no arguments on their own.

We do need the inverse mapping only for type constructors of higher ranked kind, such as Fix. The type Fix is not in the signature of *collect* and would thus not appear in a convert call for *varcollect*, but we will show the result anyway for the sake of example:

$$\begin{array}{l}
\mathsf{convert}(\mathit{collect}, \mathit{varcollect}, \mathrm{Fix}, \mathit{collect} \mapsto \mathit{varcollect}) \equiv \\
\quad \mathit{varcollect}\ \langle \mathrm{Fix}\ (\alpha :: * \to *)\rangle = \\
\qquad\qquad\qquad\qquad \begin{array}{ll}
\mathbf{letsim} & \mathit{collect}\ \langle \alpha\ (\gamma :: *)\rangle = \\
 & \qquad \begin{array}{ll} \mathbf{letsim} & \mathit{varcollect}\ \langle\gamma\rangle = \mathit{collect}\ \langle\gamma\rangle \\ \mathbf{in} & \mathit{varcollect}\ \langle \alpha\ \gamma\rangle \end{array} \\
\mathbf{in} & \mathit{collect}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle\ .
\end{array}
\end{array}$$

Cases for type constructors of higher ranked kinds appear rarely in type-indexed functions, therefore we do not require the remapping of dependencies to be invertible, and fail only lazily during the translation, i.e., only if the inverse is requested at a dependency for which it cannot be produced (this can happen because no name is mapped to that dependency, or because multiple names are mapped to that dependency).

14.5 Typing default cases

In the previous section, we have shown how a function that has a default case is type checked: all arms that are explicitly defined are type checked against the type signature of the function as usual; for the default case, the conversions are generated, and then also type checked against the type signature of the function.

We do not type check the **extends** construct directly, but rather check its translation. The reasoning behind this is relatively simple: we want to be as liberal as possible. In the general case, however, a type-indexed function is very sensitive to any change in its type.

There are several situations in which it is desirable to be able to use a different type in the derived function than in the original function. For instance, the function *collect* is polymorphic in the element type of the resulting list. That is possible because *collect*, without local redefinition or an extension via a default case, returns always the empty list. But surely, we do not want extensions to suffer from the same restriction, and indeed, *varcollect* and *termcollect* return a list of type Var.

Other possibilities are that one of the newly defined arms of the function introduces an additional dependency that the original function does not have. Or the original function has multiple generic slots, and we want the new function to have fewer. For instance, if we do not need or even do not want the possibility of function *map* to possibly change the type of its argument, we could want to restrict it to the type of *gid* in an extension. Both these changes are critical, though, if the function should be defined for type patterns involving higher-ranked type constructors. We have discussed in Section 13.3 that reducing the number of generic type variables can prove fatal in such a situation. Introducing a new dependency is not much different from that, because it implies that the function new, which remaps dependencies of the old function to the new function, is not invertible. Still, if there are no such cases in the extended function, such modifications in the types may be possible.

Instead of designing cunning type checking rules for an **extends** construct that grant this flexibility, we choose the pragmatic solution to simply type check the expansion of the **extends** construct. This is not as bad as it sounds: the expansion depends on the signature and the type signature of the original function, but not on its implementation.

While we are at discussing types of default cases, let us point out that it may be desirable to extend the function convert in such a way that not only remapping of dependencies can be performed, but also arguments can be introduced or eliminated. For instance, in the paper introducing Dependency-style Generic Haskell (Löh *et al.* 2003), a type-indexed function *update* is used to increase the salary of all employees in an organization. This function is defined using a default case as follows

$$\begin{aligned}
&\mathit{update}\ \langle a :: *\rangle :: (\mathit{update}\ \langle a\rangle) \Rightarrow \mathrm{Float} \to a \to a\\
&\mathit{update}\ \mathit{frac}\ \mathbf{extends}\ \mathit{map}\\
&\mathit{update}\ \langle \mathit{Salary}\rangle\ \mathit{frac}\ (\mathit{SalaryC}\ s) = \mathit{SalaryC}\ (s\ \text{`}\mathit{times}\text{`}\ (1 + \mathit{frac}))\ \ .
\end{aligned}$$

The function is based on *map*, because it traverses a value of the several data structures that are used to represent the organization, and produces a new value of the same type. We use only one generic type argument, because we are not modifying types anywhere. The only position where we actually change something is whenever we encounter a value of the type *Salary*, defined as

$$\mathbf{data}\ \mathit{Salary} = \mathit{SalaryC}\ \mathrm{Float}\ \ .$$

In such a case, we increase the salary by a given fraction *frac*. The argument *frac* is an argument of *update*, but not of *map*. The **extends** construct declares that we intend to introduce the variable *frac* while extending. The expansion of the default case is supposed to look as follows:

$$\begin{aligned}
&\mathit{update}\ \langle \mathrm{Int}\rangle && \mathit{frac} = \mathit{map}\ \langle \mathrm{Int}\rangle\\
&\mathit{update}\ \langle \mathrm{Char}\rangle && \mathit{frac} = \mathit{map}\ \langle \mathrm{Char}\rangle\\
&\mathit{update}\ \langle \mathrm{Float}\rangle && \mathit{frac} = \mathit{map}\ \langle \mathrm{Float}\rangle\\
&\mathit{update}\ \langle \mathrm{Unit}\rangle && \mathit{frac} = \mathit{map}\ \langle \mathrm{Unit}\rangle\\
&\mathit{update}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle && \mathit{frac} = \mathbf{let}\ \mathit{map}\ \langle \alpha\rangle = \mathit{update}\ \langle \alpha\rangle\ \mathit{frac}\\
& && \phantom{\mathit{frac} = \mathbf{let}\ }\mathit{map}\ \langle \beta\rangle = \mathit{update}\ \langle \beta\rangle\ \mathit{frac}\\
& && \phantom{\mathit{frac} = }\mathbf{in}\ \ \mathit{map}\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathit{update}\ \langle \mathrm{Prod}\ \alpha\ \beta\rangle\ \mathit{frac} = \mathbf{let}\ \mathit{map}\ \langle\alpha\rangle = \mathit{update}\ \langle\alpha\rangle\ \mathit{frac}\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathit{map}\ \langle\beta\rangle = \mathit{update}\ \langle\beta\rangle\ \mathit{frac}\\
&\qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ \ \mathit{map}\ \langle \mathrm{Prod}\ \alpha\ \beta\rangle
\end{aligned}$$

The variable *frac* is automatically threaded through the computation and respected while remapping *map* dependencies to *update*. The full **extends** statement for *update* – including a **where** clause – could be written

$$\begin{aligned}
&\mathit{update}\ \mathit{frac}\ \mathbf{extends}\ \mathit{map}\\
&\quad \mathbf{where}\ \mathit{map}\ \mathbf{as}\ \mathit{update}\ \mathit{frac}\ \ .
\end{aligned}$$

Of course, if such constructions are allowed, it becomes even harder to properly invert the dependency remapping function new. Not only the association between function names, but also the connected operations, such as the threading of *frac* in this case, must be inverted. It is thus difficult to make such extensions possible on functions that involve patterns with higher ranked type constructors. But such functions are rare.

15 Type, Newtype, Data

The Haskell language offers three constructs to define types: next to **data**, the general construct to define new datatypes that is also part of all languages discussed in this thesis, there is **newtype**, a limited version of **data** that can be used to declare a new, distinct, datatype that is isomorphic to an already existing type, and **type**, a construct to define type synonyms, which are symbolic names that can be used as abbreviations for existing types, but are not considered distinct.

In this short chapter, we will discuss how a language with generic functions can be extended with both **newtype** and **type** statements, and what effect the presence of these additional constructs has on type-indexed functions. In Figure 15.1, the relevant additional syntax is shown. The language that has all three type-definition constructs available is called FCRT, and all languages based thereon do also get an FCRT prefix. For instance, the language including generic functions, generic abstraction, and default cases on the basis of the full diversity of type-level declarations is called FCRT+gf+gabs+dc.

Type declarations

$$
\begin{array}{lll}
D & ::= & \textbf{data}\ T = \{\Lambda a_i :: \kappa_i.\}^{i \in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m} \\
 & & \qquad \text{datatype declaration} \\
 & | & \textbf{newtype}\ T = \{\Lambda a_i :: \kappa_i.\}^{i \in 1..\ell}\ C\ t \\
 & & \qquad \text{newtype declaration} \\
 & | & \textbf{type}\ T = \{\Lambda a_i :: \kappa_i.\}^{i \in 1..\ell}\ t \\
 & & \qquad \text{type synonym declaration}
\end{array}
$$

Figure 15.1: Full syntax of type declarations for language FCRT, extends Figures 3.1 and 3.12

15.1 Datatype renamings

The Haskell Report (Peyton Jones 2003) calls the functionality that is offered by the **newtype** construct the "renaming" of an existing datatype. The syntactic difference between a **newtype** and a **data** is that a **newtype** is restricted to exactly one constructor with exactly one field. We therefore use the following syntax for **newtype**:

$$\textbf{newtype}\ T = \{\Lambda a_i :: \kappa_i.\}^{i \in 1..\ell}\ C\ t\ \ .$$

In Haskell, there is a subtle difference between a **newtype** statement and a **data** statement of the same form, as for example

$$
\begin{array}{ll}
\textbf{data} & \mathit{RoomNumber} = \mathit{Room}\ \mathrm{Int} \\
\textbf{newtype} & \mathit{RoomNumber} = \mathit{Room}\ \mathrm{Int}\ \ .
\end{array}
$$

For the **data** statement, the values *Room* $\perp$ and $\perp$ are different, whereas for the **newtype** statement, they are considered the same. As a consequence, there are slightly different pattern matching rules for constructors defined via **newtype**.

Apart from that, and especially as far as type-indexed functions are considered, any **newtype** statement

$$\textbf{newtype}\ T = \{\Lambda a_i :: \kappa_i.\}^{i \in 1..\ell}\ C\ t$$

can be treated exactly as the corresponding **data** statement

$$\textbf{data}\qquad T = \{\Lambda a_i :: \kappa_i.\}^{i \in 1..\ell}\ C\ t\ \ .$$

In particular, one can define arms of type-indexed functions for datatypes defined in **newtype** statements, and types defined via **newtype** do also have structural representations, embedding-projection pairs and may occur in type arguments.

15.2 Type synonym declarations

A type synonym introduces a new name for a type expression. Type synonyms can be parametrized. Other than the **data** and **newtype** constructs, the new type names are not considered to be distinct types. They are rather alternative names for already existing types.

Furthermore, Haskell imposes some severe restrictions on the use of type synonyms in programs: they must always appear fully applied, and they must not be (directly or indirectly) recursive. They also cannot be used in **instance** declarations. All these restrictions together ensure that type synonyms can be expanded to their definitions, and can be seen as no more than a way to make the program more readable.

As such, type synonyms also pose no problems in a language that allows generic programming. The syntax for type synonyms is

$$\textbf{type}\ T = \{\Lambda a_i :: \kappa_i.\}^{i \in 1..\ell}\ t\ .$$

The type t that occurs on the right hand side is *not* required to be of kind $*$. It is thus possible to define synonyms for higher kinds without eta-expansion, i.e., we can define

$$\textbf{type}\ \mathit{AltMaybe} \quad = \text{Sum Unit}$$

instead of

$$\textbf{type}\ \mathit{AltMaybe}'\ a = \text{Sum Unit}\ a\ ,$$

which is valuable, because the synonym *AltMaybe* can still be used on its own, whereas *AltMaybe′* must always be applied to its argument.

Type synonyms can be used in type arguments that are part of generic applications, as long as long as they are fully applied. The type synonyms can then be replaced by their definitions, and specialization can proceed as usual.

A more problematic question is whether type synonyms could or should be allowed in type *patterns* in definitions of type-indexed functions. In the following, we will outline a simple way to let the answer to this question be “yes”, and outline the implications.

We want to allow type patterns of the form $T\ \{\alpha_i\}^{i \in 1..\ell}$, where the pattern can be checked to be of kind $*$, and T is the name of a type synonym. The type requirements for an arm associated with such a pattern are exactly as for other arms. The interesting part is to analyze when such an arm should be applied.

We take a very simplistic approach here. Type synonyms can occur in the signatures of type-indexed functions as any other type. They are not expanded if they occur in type patterns. We do also not expand type synonyms that occur in type arguments, but treat them as other named types and specialize as usual.

During specialization, we can thus reach a situation where we need a component $\mathtt{cp}(x, T)$ for some generic function x on type synonym T. Now, if T is in the signature of x, we take the explicitly defined $\mathtt{cp}(x, T)$. Otherwise, we try to generate it. For datatypes, generation of new components proceeds as explained in Chapter 11, making use of the structural representation of the datatype instead of the original datatype and using the lifted embedding-projection pair to map between the two components that work on different, but isomorphic datatypes.

For type synonyms, we can reuse the existing mechanism if we can define a structural representation, plus embedding-projection pairs. This is trivial, as we can simply let the right hand side of a type synonym double as its structural representation, together with the identity embedding-projection pair *EP id id*.

For example, for strings, that in Haskell are defined as a type synonym

$$\textbf{type}\ \text{String} = [\text{Char}]\ ,$$

we would define

$$\begin{aligned}&\mathtt{Str}(\text{String}) \equiv [\text{Char}]\\&\mathtt{ep}(\text{String}) = \mathit{EP}\ \mathit{id}\ \mathit{id}\ ,\end{aligned}$$

and then reuse the established machinery to treat generic applications involving String. We could then define a special arm of *showP* (cf. Section 17.1.2) for strings, as

$$\mathit{showP}\ \langle\text{String}\rangle\ p\ \mathit{xs} = \texttt{"\textbackslash""} +\!\!+\ \mathit{map}\ \langle[\,]\rangle\ (\mathit{show}\ \langle\text{Char}\rangle)\ \mathit{xs} +\!\!+ \texttt{"\textbackslash""}\ .$$

With this extra case, the function *show* can be used as follows

$$\begin{aligned}&\mathit{show}\ \langle\text{String}\rangle\ \ \texttt{"Hello"}\\&\mathit{show}\ \langle[\text{Int}]\rangle\quad [1,2,3,4,5]\\&\mathit{show}\ \langle[\text{Char}]\rangle\ \texttt{"Hello"}\end{aligned}$$

to produce the results

```
"\"Hello\""
"[1, 2, 3, 4, 5]"
"['H', 'e', 'l', 'l', 'o']",
```

where we assume that *showP* has components for both Char and Int that implement its functionality in the same way as Haskell's standard *show* function. The first string is shown as a string, because the type argument explicitly mentions

String, whereas the two others are shown as normal lists, using the [ ] arm that *showP* provides.

Note that the only difference between the first and the third case is the way in which the type argument has been written! We do not make any attempts to find out that [Char] is the same as String. We also respect the use (or non-use) of type synonyms in the definitions of other types. In the presence of the definitions

> **newtype** $Name_1 = N_1$ String
> **newtype** $Name_2 = N_2$ [Char] ,

the expression

> (*show* $\langle Name_1\rangle$ `"foo"`, *show* $\langle Name_2\rangle$ `"bar"`)

evaluates to

```
("\"foo\"","['b', 'a', 'r']") .
```

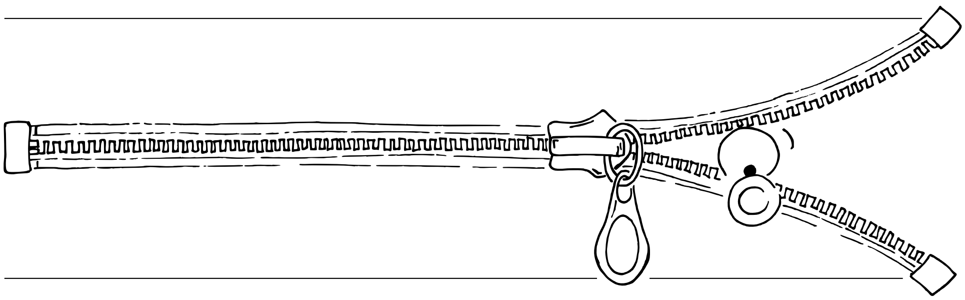
Being able to treat type synonyms in a special way in type-indexed function definitions can turn out to be both a blessing and a curse.

A blessing, because defining a type synonym and associated special behaviour of a type-indexed function can be a very lightweight and readable way to modify the normal, generically defined behaviour that would be derived for a datatype. A programmer can choose names for type synonyms that cover the intention in which a value of this type should be used. Achieving exactly the same using local redefinition can turn out to be much harder.

A curse, because a type-indexed function can have different results for two calls $x\ \langle t_1\rangle$ and $x\ \langle t_2\rangle$, even though t_1 and t_2 are considered to be the same types in Haskell. This makes reasoning about calls to type-indexed functions much harder, and can also lead to surprising results for users who do not expect that the behaviour of a program can change if they just replace a type expression by an equivalent type synonym somewhere.

Furthermore, we have to watch out if we perform type argument inference, as described in Section 13.1. If the presence of type synonyms can influence the result of a generic application, it is required that it is specified when and where an inferred type argument will make use of type synonyms. A simple solution here is to demand that an inferred type argument is always fully expanded, thus free of any type synonyms. If that does not yield the behaviour that is desired, one can always override the inferred type argument by specifying it explicitly.

16 Type-indexed Datatypes



Type-indexed functions are functions that are parametrized by a type argument, and have access to the structure of the type argument during definition, such that they can behave differently for different types. If defined for the types and type constructors Unit, Sum, Prod, and Zero, that make up the standard view on datatypes (cf. Chapter 10), type-indexed functions become generic, and work for nearly all Haskell datatypes.

The same mechanism can be useful on the type level: a **type-indexed datatype** is a datatype with a type argument, and we can pattern match on the type argument, making use of its structure while defining the datatype. A type-indexed datatype can thus have a different implementation depending on the type argument. Type-indexed datatypes can also be *generic* – if they are defined for the suitable type constructors, an implementation can be derived for a large class of Haskell datatypes.

In Section 16.1, we will show how to define a type-indexed datatype by means of an example. We will also discuss how such a datatype can be put to use, mainly in generic functions. In Sections 16.2 and 16.3, we discuss – still using the example – some peculiarities of type-indexed datatypes and how they can be translated.

After that, in Sections 16.4 and 16.5, we will analyze how some of the concepts known from type-indexed and generic functions can be transferred to the type domain. In Section 16.6, we discuss the "zipper", another example of a type-indexed datatype. Section 16.7 contains a discussion the language extensions necessary to support type-indexed and generic datatypes.

16.1 Type-indexed tries

A digital search tree or **trie** is a search tree scheme that employs the structure of search keys to organize information efficiently. Searching is useful for various datatypes, so we would like to allow both keys and information to be of any datatype.

If the key type should be flexible, and the structure of the keys should be used in the organization of the data structure, then the logical consequence is that we need a type-indexed – or even better, generic – datatype.

Defining a type-indexed datatype is not much different from the definition of a type-indexed function: we define several cases of a datatype, using different type patterns. Let us call the datatype of type-indexed tries FMap. It is defined as follows:

$$
\begin{array}{ll}
\textbf{type}\ \mathrm{FMap}\ \langle\mathrm{Int}\rangle & v = \mathrm{IntMap}\ v \\
\textbf{type}\ \mathrm{FMap}\ \langle\mathrm{Char}\rangle & v = \mathrm{CharMap}\ v \\
\textbf{type}\ \mathrm{FMap}\ \langle\mathrm{Unit}\rangle & v = \mathrm{Maybe}\ v \\
\textbf{type}\ \mathrm{FMap}\ \langle\mathrm{Sum}\ \alpha\ \beta\rangle & v = (\mathrm{FMap}\ \langle\alpha\rangle\ v, \mathrm{FMap}\ \langle\beta\rangle\ v) \\
\textbf{type}\ \mathrm{FMap}\ \langle\mathrm{Prod}\ \alpha\ \beta\rangle & v = \mathrm{FMap}\ \langle\alpha\rangle\ (\mathrm{FMap}\ \langle\beta\rangle\ v)\ .
\end{array}
$$

All cases of the type definition start with the keyword **type**. This indicates that each of the cases behaves as if it were a type synonym. Therefore, the right hand side is simply a type expression. It is also possible to use **newtype** and **data** constructs in the definition of a type-indexed type, and even to use different constructs for different cases. The right hand side of a case always reflects the construct that is used to define that particular case.

Each of the cases above takes a type parameter v, for the type of values that is stored in the trie. For integers and characters, we assume that we have predefined finite maps, called IntMap and CharMap. A not very efficient, but working, possibility to implement these would be as lists of pairs, $[(\mathrm{Int}, v)]$ or $[(\mathrm{Char}, v)]$, respectively, but we could, of course, choose another, more ingenious definition.

A trie for the Unit type is simply Maybe v. We use *Nothing* to denote the empty trie in this case, where the key *Unit* has no associated value. Otherwise, we store the associated value x as *Just* x.

A trie for a Sum datatype is a pair of two tries, for each of the components. In the left trie, we store entries of which the keys are of the form *Inl x*, and in the right trie those whose keys are of the form *Inr x*.

For the Prod datatype, we use a nested trie to store entries. The outer trie has key values that correspond to the left component of the Prod values. Each of these keys is associated with another trie, that maps right components of Prod values to values of type v.

The definition of the FMap datatype corresponds to the laws of exponentials. If we denote the type of finite maps from keys of type t to values of type u by $t \rightarrow_{\text{fin}} u$, use 1 for the Unit type and use the infix operators $+$ and $\times$ for the Sum and Prod type constructors, we have the following isomorphisms:

$$\begin{aligned} 1 \rightarrow_{\text{fin}} u &\cong u \\ (t_1 + t_2) \rightarrow_{\text{fin}} u &\cong t_1 \rightarrow_{\text{fin}} u \times t_2 \rightarrow_{\text{fin}} u \\ (t_1 \times t_2) \rightarrow_{\text{fin}} u &\cong t_1 \rightarrow_{\text{fin}} (t_2 \rightarrow_{\text{fin}} u) \ . \end{aligned}$$

The only difference is that we use Maybe u instead of u in the case for Unit. While the isomorphisms are valid for total finite maps, we allow partial finite maps in our definition of FMap.

Having defined FMap for Unit, Sum, and Prod makes it *generic* in much the same way as a generic function: we can apply FMap not only to type terms that are built from the type constructors appearing in the *signature*, but to all datatypes of which the translation into the standard view is expressible using these datatypes.

The definition of FMap makes use of dependency variables, and it calls FMap recursively on these dependency variables on the right hand side of the definition. For type-indexed functions, such behaviour causes a dependency that is recorded in the type signature. For type-indexed types, we also keep track of dependencies, but they are stored in the *kind signature* of the type. The kind signature of FMap is:

$$\text{FMap}\ \langle a :: * \rangle :: (\text{FMap}) \Rightarrow * \rightarrow * \ .$$

The name of the type variable a is irrelevant. We mention it only in order to stay as close as possible to the type signatures of type-indexed functions; but unlike the situation for type-indexed functions, the variable cannot occur on the right hand side of the signature. The right hand side is structured similar to a type signature for a type-indexed function: the dependencies are listed – in this case only FMap itself – before the kind, which in this case is $* \rightarrow *$, because all arms of the type are parametrized over the type of values that are to be stored in the finite map.

The notation for dependencies corresponds to the simple case for type-indexed functions that we have discussed in Chapter 6. Since neither kind variables nor polymorphic kinds exist in our language, the kind of a type-indexed type is always constant. There is thus just one way to depend on another type-indexed type. Consequently, there is no need to parametrize a kind signature over a type tuple, or to add type tuples to the dependencies.

Nevertheless, the form of the kind signature implies that we have *qualified kinds* in the presence of type-indexed types. Using an algorithm corresponding to gapp (the simple variant has been described in Section 6.3.2) on the type level, we can assign a qualified kind to each call of a type-indexed type. Figure 16.1 shows examples for type arguments of different form. In all four cases, A is assumed to be free of dependency variables. The case for $A\ (\alpha :: *)\ (\beta :: *)$, for example, tells

$$
\begin{array}{ll}
\mathrm{FMap}\ \langle A :: * \rangle & :: * \to * \\
\mathrm{FMap}\ \langle A\ (\alpha :: *) :: * \rangle & :: (\mathrm{FMap}\ \langle \alpha \rangle :: * \to *) \Rightarrow * \to * \\
\mathrm{FMap}\ \langle A\ (\alpha :: *)\ (\beta :: *) :: * \rangle & :: (\mathrm{FMap}\ \langle \alpha \rangle :: * \to *, \mathrm{FMap}\ \langle \beta \rangle :: * \to *) \\
 & \quad \Rightarrow * \to * \\
\mathrm{FMap}\ \langle A\ (\alpha :: * \to *) :: * \rangle & :: (\mathrm{FMap}\ \langle \alpha\ \gamma \rangle :: (\mathrm{FMap}\ \langle \gamma \rangle :: * \to *) \\
 & \qquad\qquad\qquad\qquad \Rightarrow * \to *) \\
 & \quad \Rightarrow * \to *
\end{array}
$$

Figure 16.1: Kinds for generic applications of FMap to type arguments of different form

us that while defining FMap $\langle \mathrm{Sum}\ \alpha\ \beta \rangle$ or FMap $\langle \mathrm{Prod}\ \alpha\ \beta \rangle$, we may assume on the right hand side that FMap $\langle \alpha \rangle$ and FMap $\langle \beta \rangle$ are both available with kind $* \to *$. Analogous to qualified types, we will translate qualified kinds into functional kinds with explicit arguments.

The natural way to define operations on type-indexed datatypes is to use type-indexed functions. We need to distinguish cases based on the type argument, because each time we have to deal with a different data structure. The function *lookup* takes a key and a trie and returns a Maybe containing either the value that is associated with the key in the trie or *Nothing* if no value is associated with the key:

$$
\begin{array}{llll}
\mathit{lookup}\ \langle \mathrm{Unit} \rangle & \mathit{Unit} & z & = z \\
\mathit{lookup}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inl}\ x) & (z_1, z_2) & = \mathit{lookup}\ \langle \alpha \rangle\ x\ z_1 \\
\mathit{lookup}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & (\mathit{Inr}\ x) & (z_1, z_2) & = \mathit{lookup}\ \langle \beta \rangle\ x\ z_2 \\
\mathit{lookup}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle & (x \times y)\ z_1 & & = \mathbf{case}\ \mathit{lookup}\ \langle \alpha \rangle\ x\ z_1\ \mathbf{of} \\
 & & & \qquad \mathit{Nothing} \to \mathit{Nothing} \\
 & & & \qquad \mathit{Just}\ z_2 \quad \to \mathit{lookup}\ \langle \beta \rangle\ y\ z_2 \ .
\end{array}
$$

We omit the cases for Int and Char, which deal with the primitive, abstract finite maps IntMap and CharMap. The definition is straightforward if one bears in mind the idea of the definition of the type-indexed trie. The second argument is the trie, therefore it is of different form in each of the three arms: a Maybe v, where v is the value type, for Unit, a pair in the Sum case, and a nested finite map for Prod. If we want to lookup *Unit* in a unit trie, we can just return the Maybe as the result. In the case Sum α β, we recursively call *lookup*, either on α or on β, depending on whether the key is constructed by *Inl* or *Inr*. In the Prod case, we take the nested trie z_1 and look for the first component of the key x, in that trie. If this key is not contained in the trie, then we return *Nothing*. If it is, we end up with another trie z_2, in which we can look for the second component of the key y.

The type signature of *lookup* reflects the use of a type-indexed type:

$$\mathit{lookup}\ \langle a :: *\rangle :: (\mathit{lookup}\ \langle a\rangle) \Rightarrow \forall v :: *.\ a \to \text{FMap}\ \langle a\rangle\ v \to \text{Maybe}\ v\ \ .$$

Note that in this case, we really need an inner quantification for the type variable v. A type signature of the form

$$\mathit{lookup}\ \langle a :: * \mid v :: *\rangle :: \ \ldots$$

would not do! The reason is that *lookup* is polymorphically recursive in the Prod arm: *lookup* $\langle\alpha\rangle$ uses the function with result type Maybe (FMap $\langle\beta\rangle$ v), whereas *lookup* $\langle\beta\rangle$ uses it with Maybe v as result type.

Another example of a type-indexed function on tries is the *empty* function that produces an empty trie:

$$\begin{aligned}
&\mathit{empty}\ \langle\text{Unit}\rangle && = \mathit{Nothing}\\
&\mathit{empty}\ \langle\text{Sum}\ \alpha\ \beta\rangle && = (\mathit{empty}\ \langle\alpha\rangle, \mathit{empty}\ \langle\beta\rangle)\\
&\mathit{empty}\ \langle\text{Prod}\ \alpha\ \beta\rangle && = \mathit{empty}\ \langle\alpha\rangle\ \ .
\end{aligned}$$

For Unit, we return *Nothing*. For a Sum, we return a pair of empty tries. For Prod, an empty trie on α suffices. The type signature of *empty* is:

$$\mathit{empty}\ \langle a :: *\rangle :: (\mathit{empty}\ \langle a\rangle) \Rightarrow \forall v :: *.\ \text{FMap}\ \langle a\rangle\ v\ \ .$$

Apart from the type argument, *empty* takes no further arguments. Like *enum* or *card*, it is a type-indexed value.

All other operations on tries, such as *insert* and *delete*, can also be defined as generic functions. Therefore, FMap can be treated as an abstract datatype only accessible through these functions. As already said, it is rare that one feels the need to define non-generic functions that use specific instances of type-indexed datatypes. In such a situation it is necessary that we know more about how type-indexed datatypes are specialized.

16.2 Explicit specialization

Other than type-indexed functions, type-indexed datatypes have to be specialized explicitly. This is because a choice has to be made whether they should be specialized using a **type** or a **newtype** construct, a choice that is hard to make optimally using only the compiler. Another reason will become apparent when we discuss modules in Chapter 18.

Type indexed datatypes are, similarly to functions, specialized to an application of several components, thus constructor names can soon prevail and obfuscate the code. While Haskell type synonyms have the advantage of not introducing constructors, they are also subject to restrictions: they may not be recursive (not even indirectly, except if a proper datatype intervenes), and they must always be fully applied.

Because of these restrictions, not all specializations of type-indexed datatypes can be made using type synonyms only. For example, the specialization to a type argument that involves a recursive datatype will always be recursive and must thus always involve a **newtype**. For systems of mutually recursive datatypes, it is necessary to use a **newtype** somewhere, but not necessarily everywhere. Instead of equipping the compiler with some heuristic that tries to determine when to choose a **newtype** and when a **type**, we leave this choice to the user, and require the user to explicitly make this choice before using the type.

Explicit specialization means that we cannot use a call such as *lookup* ⟨[Int]⟩ without preparation. Such a call would result in a specialization error, because *lookup* ⟨[Int]⟩ requires an FMap ⟨[Int]⟩. Whereas FMap ⟨Int⟩ is directly available as a case of the definition, FMap cannot be automatically specialized to the list type constructor [ ]. The type constructor [ ] is not in the signature of FMap, and the compiler has to be instructed whether to generate the component using a **newtype** or a **type** construct.

If we first write

newtype FMap ⟨[ ]⟩ **as** *FMapList* ,

where *FMapList* is a constructor name, the call *lookup* ⟨[Int]⟩ can be specialized successfully. Such a construct is called a **specialization request**, and here asks the compiler to specialize FMap to the list type using a **newtype** construct with the constructor *FMapList*. As discussed above, we cannot use a type synonym in this place because [ ] is a recursive datatype. An example for a specialization request for a type synonym is

type FMap ⟨Bool⟩ .

Only **newtype** and **type** statements are valid as specialization requests, because the additional flexibility of the **data** construct is of no use here. The type argument of a specialization request must be a single named type, or a (possibly nested) type application to a named type. The latter occurs if a type-indexed datatype is to be used on a type argument that contains a type-indexed datatype again, such as in the call FMap ⟨FMap ⟨[Int]⟩ Char⟩, that describes the type of finite maps which have keys that are finite maps from lists of integers to characters. We then need a specialization request such as

newtype FMap ⟨FMap ⟨[ ]⟩⟩ **as** *FMapFMapList*

in addition to the specialization request for FMap to [ ] (see also Section 16.8).

16.3 Idea of the translation

In this section, we will briefly sketch how the FMap example is translated, before formally discussing the process in Section 16.7.

For each of the cases in the FMap definition, a *component* is generated in the output program, only that the component is now a type declaration instead of a function declaration:

$$
\begin{array}{ll}
\textbf{type}\ \mathsf{Cp}(\text{FMap}, \text{Int}) & v = \text{IntMap}\ v \\
\textbf{type}\ \mathsf{Cp}(\text{FMap}, \text{Char}) & v = \text{CharMap}\ v \\
\textbf{type}\ \mathsf{Cp}(\text{FMap}, \text{Unit}) & v = \text{Maybe}\ v \\
\textbf{type}\ \mathsf{Cp}(\text{FMap}, \text{Sum})\ \mathsf{Cp}(\text{FMap}, \alpha)\ \mathsf{Cp}(\text{FMap}, \beta)\ v = & \\
\qquad (\mathsf{Cp}(\text{FMap}, \alpha)\ v, \mathsf{Cp}(\text{FMap}, \beta)\ v) & \\
\textbf{type}\ \mathsf{Cp}(\text{FMap}, \text{Prod})\ \mathsf{Cp}(\text{FMap}, \alpha)\ \mathsf{Cp}(\text{FMap}, \beta)\ v = & \\
\qquad \mathsf{Cp}(\text{FMap}, \alpha)\ (\mathsf{Cp}(\text{FMap}, \beta)\ v)\ . &
\end{array}
$$

The internal operation Cp is the type level analogue to cp. It produces a named type if applied to two named types, but a type variable if applied to a named type and a dependency variable, as in $\mathsf{Cp}(\text{FMap}, \alpha)$. The definition of FMap depends on itself, hence the kind signature

$$\text{FMap}\ \langle a :: * \rangle :: (\text{FMap}) \Rightarrow * \rightarrow *\ .$$

In the translation, we can see this fact reflected in the presence of two type arguments for the dependencies that are provided to the Sum and Prod components.

A direct consequence of this translation scheme is that a reference to the type FMap ⟨[Int]⟩ ends up as a reference to the type $\mathsf{Cp}(\text{FMap}, [\,])\ \mathsf{Cp}(\text{FMap}, \text{Int})$ in the resulting program. Such type applications in type arguments are always simplified, and components are defined for a type-indexed type applied to a single named type.

A specialization request such as

$$\textbf{newtype}\ \text{FMap}\ \langle[\alpha]\rangle\ \textbf{as}\ \textit{FMapList}$$

leads to the generation of a component of FMap for the list type constructor. Because FMap is not defined for lists, we make use of the genericity of FMap and structurally represent the list datatype, much as we do for generic functions. Recall that

$$\textbf{type}\ \mathsf{Str}([\,])\ (a :: *) = \text{Sum Unit}\ (\text{Prod}\ a\ [a])\ ,$$

We can specialize type-indexed types to type terms in analogy to how we specialize type-indexed functions to type terms. Thus, we can translate FMap $\langle\mathsf{Str}([\,])\rangle$ to

$$\begin{aligned}
&\textbf{type}\ \mathsf{Cp}(\text{FMap}, \mathsf{Str}([\,]))\ (\mathsf{Cp}(\text{FMap}, \alpha) :: *) = \\
&\quad \mathsf{Cp}(\text{FMap}, \text{Sum}) \\
&\qquad \mathsf{Cp}(\text{FMap}, \text{Unit}) \\
&\qquad \big(\mathsf{Cp}(\text{FMap}, \text{Prod})\ \mathsf{Cp}(\text{FMap}, \alpha) \\
&\qquad\qquad\qquad\qquad (\mathsf{Cp}(\text{FMap}, [\,])\ \mathsf{Cp}(\text{FMap}, \alpha))\big)\ .
\end{aligned}$$

We define a type synonym $\mathsf{Cp}(\text{FMap}, \mathsf{Str}([\,]))$ to hold the translation. For type-indexed functions, we subsequently needed a wrapper to convert the component for the structural representation type into a component for the original type. The reason is that we were defining two functions, one of which referred to the structural representation type whereas the other had to refer to the original type. The types of the components were dictated by the type signature of the function.

Here, the situation is much simpler! We define types, and their kinds are dictated by the kind signature of the type indexed type, and the base kind of a function is constant, i.e., it does not involve the type argument in any way. As a consequence, the above component for the structural representation has the same kind as $\mathsf{Cp}(\text{FMap}, [\,])$, and we can almost use it as it stands as the component for lists itself. The only thing we have to add is a **newtype** construct with the constructor that has been requested:

$$\begin{aligned}
&\textbf{newtype}\ \mathsf{Cp}(\text{FMap}, [\,])\ (\mathsf{Cp}(\text{FMap}, \alpha) :: *) = \\
&\quad \textit{FMapList}\ (\mathsf{Cp}(\text{FMap}, \mathsf{Str}([\,]))\ \mathsf{Cp}(\text{FMap}, \alpha))\ .
\end{aligned}$$

In addition, we generate another embedding-projection pair:

$$\mathsf{ep}(\mathsf{Cp}(\text{FMap}, [\,])) :: \forall a :: *.\ \text{EP}\ (\mathsf{Cp}(\text{FMap}, [\,])\ a)\ (\mathsf{Cp}(\text{FMap}, \mathsf{Str}([\,]))\ a)$$

$$\begin{aligned}\mathsf{ep}(\mathsf{Cp}(\text{FMap},[\,])) = \mathbf{let}\; & \textit{from}\ (\textit{FMapList}\ x) = x\\ & \textit{to}\quad x \qquad\qquad = \textit{FMapList}\ x\\ \mathbf{in}\;\; & \textit{EP from to}\ .\end{aligned}$$

We need these conversion functions when translating type-indexed functions that make use of type-indexed datatypes.

For a specialization request to a type synonym, such as

$$\mathbf{type}\ \text{FMap}\ \langle\text{Bool}\rangle\ ,$$

we have to perform even less work. The component for the structural representation is in this case

$$\begin{aligned}&\mathbf{type}\ \mathsf{Cp}(\text{FMap},\mathsf{Str}(\text{Bool})) =\\ &\quad \mathsf{Cp}(\text{FMap},\text{Sum})\ \mathsf{Cp}(\text{FMap},\text{Unit})\ \mathsf{Cp}(\text{FMap},\text{Unit})\ ,\end{aligned}$$

and it can directly be used as the component for the original type, which in this case is also a type synonym:

$$\mathbf{type}\ \mathsf{Cp}(\text{FMap},\text{Bool}) = \mathsf{Cp}(\text{FMap},\mathsf{Str}(\text{Bool}))\ .$$

Hence, the embedding-projection pair in this case is the identity pair *EP id id*.

If we want to translate type-indexed functions on type-indexed datatypes, we can proceed almost as before. For example, the specialization of call *lookup* $\langle[\text{Int}]\rangle$ would require the component $\mathsf{cp}(\textit{lookup},[\,])$ to be generated. As before, we go via the structure type, and can define the component as a translation of a wrapper of the following form:

$$\begin{aligned}&\textit{lookup}\ \langle[\alpha]\rangle =\\ &\quad \mathbf{let}\ldots\\ &\quad \mathbf{in}\ \textit{to bimap}\ \langle\ldots\rangle\ (\textit{lookup}\ \langle\mathsf{Str}([\,])\ \alpha\rangle)\ ,\end{aligned}$$

but we have to adapt the *bimap* call slightly. Previously, we would have filled in the base type of *lookup* for the type argument, $\mathsf{mbase}(\textit{lookup}\ \langle\beta\rangle)$, the generic slot instantiated to a fresh dependency variable, and then redefined $\textit{bimap}\ \langle\beta\rangle = \mathsf{ep}([\,])$ on this dependency variable.

However, $\mathsf{mbase}(\textit{lookup}\ \langle\beta\rangle)$ is

$$\forall v :: *.\ \beta \to \text{FMap}\ \langle\beta\rangle\ v \to \text{Maybe}\ v\ .$$

Apart from the fact that there is a universally quantified type variable of kind $*$ in this type, which we can handle according to the methods in Section 11.3, a more imminent problem is that this type is not of kind $*$! The dependency variable β appears in the type argument of FMap, and FMap depends on itself. Hence, the above type has an unsatisfied dependency constraint on the kind level, and is of the qualified kind

$$(\text{FMap}\ \langle\beta\rangle :: * \to *) \Rightarrow *\ .$$

Furthermore, we should be aware that the type of *lookup* $\langle \mathsf{Str}([\,])\ \alpha\rangle$ turns out to be

$$\begin{aligned}&\forall a :: *.\ (\mathit{lookup}\ \langle\alpha\rangle :: \forall v :: *.\ a \to \text{FMap}\ \langle a\rangle\ v \to \text{Maybe}\ v)\\ &\qquad\qquad \Rightarrow \forall v :: *.\ \mathsf{Str}([\,])\ a \to \text{FMap}\ \langle \mathsf{Str}([\,])\ a\rangle\ v \to \text{Maybe}\ v\ .\end{aligned}$$

Here, the type $\mathsf{Str}([\,])$ appears within the type argument of FMap! As we have just seen, FMap $\langle\mathsf{Str}([\,])\rangle$ corresponds to $\mathsf{Cp}(\text{FMap}, \mathsf{Str}([\,]))$, which can be converted into the component for the original type $\mathsf{Cp}(\text{FMap}, [\,])$ via the embedding-projection pair $\mathsf{ep}(\mathsf{Cp}(\text{FMap}, [\,]))$.

The solution is to use local redefinition on the type level to get rid of the dependency constraint in such a way that we can use $\mathsf{ep}(\mathsf{Cp}(\text{FMap}, [\,]))$ at the position of FMap $\langle\beta\rangle$ in the base type:

```
lookup ⟨[α]⟩ =
  let bimap ⟨β⟩ = ep([ ])
      bimap ⟨γ⟩ = ep(Cp(FMap, [ ]))
  in  to bimap ⟨Let type FMap ⟨β⟩ (a :: ∗) = γ In mbase(lookup ⟨β⟩)⟩
           (lookup ⟨Str([ ]) α⟩)  .
```

We choose to redefine FMap $\langle\beta\rangle$ to another dependency variable, γ, which in turn creates another dependency of the *bimap* function, for which we can plug in the desired embedding-projection pair.

16.4 Local redefinition on the type level

What we have just seen while defining the wrapper for *lookup* $\langle[\alpha]\rangle$, is an example of local redefinition on the type level. Local redefinition works on the type level in much the same way as on the value level (cf. Chapter 8). However, local redefinition is a construct that appears within type expressions. For example, we could write

```
Let type FMap ⟨α⟩ (a :: ∗) = [a]
In  FMap ⟨Tree α⟩
```

to locally use a list of values as a finite map implementation for keys which are of the element type of the Tree.

In the *lookup* example, we have used a dependency variable on the right hand side,

> **Let type** FMap $\langle\beta\rangle$ $(a :: *) = \gamma$
> **In** $\mathsf{mbase}(\mathit{lookup}\ \langle\beta\rangle)$.

This is possible because the entire type expression is used as a type argument in this case.

Although we have used type synonyms in the declaration part of the local redefinition, it is theoretically possible to use **newtype** and **data** as well, just as all three type definition constructs may appear in the arms of a typecase.

On the value level, we translate a local redefinition by mapping it to a normal let statement in FCR (cf. Section 8.5). On the type level, however, there is no **Let**. We therefore must, in essence, add a **Let** to the type language: the translation should substitute the types declared in a **Let** within the body of the construct.

We can also lift the short notation of Section 8.1 to the type level, which allows us to write local redefinition as type application when there is only one dependency. Using short notation, we could write the first example above as

> FMap $\langle$Tree$\rangle$ [ ]

instead. Another example of local redefinition on the type level appears in Section 16.6.

16.5 Generic abstraction on the type level

Local redefinition is not the only concept from type-indexed functions that can be transferred to type-indexed datatypes: type-indexed types can also be defined via generic abstraction. For instance, if we want to fix the value type of a finite map to Int, without fixing the key type, we can define

> **type** Frequency $\langle\alpha :: *\rangle$ = FMap $\langle\alpha\rangle$ Int .

Whereas generic abstraction and local redefinition are almost indistinguishable syntactically on the value level, they are clearly separated on the type level. Type-level local redefinition is a construct in the language of types, whereas generic abstraction uses a type declaration to define a new entity. Much as a typecase-based definition, a definition via generic abstraction can also make use of all three type definition keywords – **type**, **newtype**, and **data** – with their usual differences.

16.6 The Zipper

The zipper (Huet 1997) is a data structure that is used to represent a tree together with a subtree that is the focus of attention, where that focus may move left, right, up or down in the tree. The zipper is used in applications where the user interactively manipulates trees; for instance, in editors for structured documents such as proofs or programs.

The focus of the zipper may only move to recursive components. Consider, for example, the datatype Tree:

$$\textbf{data}\ \text{Tree}\ (a :: *) = \mathit{Leaf} \mid \mathit{Node}\ (\text{Tree}\ a)\ a\ (\text{Tree}\ a)\ \ .$$

If the left subtree is the current focus, moving right means moving to the right subtree, not to the label of type a. We therefore must have access to the recursive positions in the tree, and thus decide to represent datatypes as fixpoints of their pattern functors, as we have done before in Section 12.3.

The zipper works on locations, where locations are pairs of a value of the datatype over which we navigate (the focus), together with a context (representing the rest of the structure). The type of locations is a type-indexed type, defined via generic abstraction:

$$\textbf{type}\ \text{Loc}\ \langle \gamma :: * \to * \rangle = \Big(\text{Fix}\ (\text{ID}\ \langle \gamma \rangle), \text{Context}\ \langle \gamma \rangle\ \big(\text{Fix}\ (\text{ID}\ \langle \gamma \rangle)\big)\Big)\ \ .$$

The location type is parametrized over a type of kind $* \to *$, the pattern functor of a datatype. This requires the datatype we want to navigate on to be transformed manually into such a form that it can be expressed as a fixpoint of a pattern functor. The pattern functor thus obtained for the tree type is

$$\textbf{data}\ \text{TreeF}\ (a :: *)\ (b :: *) = \mathit{LeafF} \mid \mathit{NodeF}\ b\ a\ b\ ,$$

and Tree a is isomorphic to Fix (TreeF a) (cf. Sections 12.3 and 17.2).

The type Loc depends on two other type-indexed types, ID and Context, and thus has kind

$$\text{Loc}\ \langle a :: * \to * \rangle :: (\text{Context}, \text{ID}) \Rightarrow *\ \ .$$

The type-indexed type ID is built-in and somewhat special. It is the identity type-indexed type and reduces to the type argument for all types. One can think of it as being implicitly defined as

$$\textbf{type}\ \text{ID}\ \langle T\ \{\alpha_i\}^{i \in 1..n} \rangle = T\ \{(\text{ID}\ \langle \alpha_i \rangle)\}^{i \in 1..n}$$

for all named types T, including abstract types. Another possible way to look at the ID type is as a way to convert a dependency variable into a type that can be used in an ordinary type expression.

The type of contexts Context stores a path from the root of the tree to the current point of focus, and in addition it stores the information on that path, so that we can merge the context with the point of focus to get the complete structure – hence the name "zipper" for the complete data structure.

For a functor, the generic type Ctx is defined to hold the possible paths:

$$
\begin{array}{ll}
\text{Ctx } \langle a :: * \rangle :: (\text{Ctx}, \text{ID}) \Rightarrow * & \\
\textbf{type}\ \text{Ctx } \langle \text{Int} \rangle & = \text{Zero} \\
\textbf{type}\ \text{Ctx } \langle \text{Char} \rangle & = \text{Zero} \\
\textbf{type}\ \text{Ctx } \langle \text{Unit} \rangle & = \text{Zero} \\
\textbf{type}\ \text{Ctx } \langle \text{Sum}\ \alpha\ \beta \rangle & = \text{Sum } (\text{Ctx } \langle \alpha \rangle)\ (\text{Ctx } \langle \beta \rangle) \\
\textbf{type}\ \text{Ctx } \langle \text{Prod}\ \alpha\ \beta \rangle & = \text{Sum } \big(\text{Prod } (\text{Ctx } \langle \alpha \rangle)\ (\text{ID } \langle \beta \rangle)\big) \\
 & \qquad\quad \big(\text{Prod } (\text{ID } \langle \alpha \rangle)\ (\text{Ctx } \langle \beta \rangle)\big)\ .
\end{array}
$$

This type can de seen as the derivative (as in calculus) of a datatype (McBride 2001). The real type of contexts Context is defined as the fixpoint of a variation of type Ctx, namely as

$$
\begin{array}{l}
\text{Context } \langle a :: * \to * \rangle :: (\text{Ctx}, \text{ID}) \Rightarrow * \to * \\
\textbf{type}\ \text{Context } \langle \gamma :: * \to * \rangle\ (a :: *) = \text{Fix } (\text{CtxF } \langle \gamma \rangle\ a)\ ,
\end{array}
$$

where CtxF is defined via generic abstraction from Ctx below. Because the type Context represents paths down the structure of a tree, we must provide the possibility for a path to end. Therefore, we allow an occurrence of *Stop* in each recursive position, which marks the end of the path, or *Down*, which expresses that the path continues to the next recursive layer:

$$
\begin{array}{l}
\textbf{data}\ \text{CtxF } \langle \gamma :: * \to * \rangle\ (a :: *)\ (r :: *) = \mathit{Stop} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Down}\ \big(\textbf{Let type}\ \text{Ctx } \langle \alpha \rangle = a \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{type}\ \text{ID } \langle \alpha \rangle = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{In}\ \ \text{Ctx } \langle \gamma\ \alpha \rangle\big)\ .
\end{array}
$$

Having made these preparations, we are now able to define navigation functions *up*, *down*, *left*, and *right* on locations, that move the focus to the parent node, the leftmost child, to the next child on the same level to the left, and to the next child on the same level to the right, respectively. Here, we will only define *down*. For a far more thorough treatment with more explanations and examples, consult the paper on type-indexed datatypes (Hinze *et al.* 2002).

We define *down* in terms of *first*, which tries to determine the leftmost recursive child in the tree. We simulate a function that is indexed over functors of kind $* \to *$, in the same way as described in Section 12.5:

```
first' ⟨a :: ∗⟩ :: (first' ⟨a⟩) ⇒ ∀b :: ∗. a → Maybe (b, Ctx ⟨a⟩)
first' ⟨Int⟩        x         = Nothing
first' ⟨Char⟩       x         = Nothing
first' ⟨Unit⟩       Unit      = Nothing
first' ⟨Sum α β⟩    (Inl x)   = do (t, cx) ← first' ⟨α⟩ x
                                    return (t, Inl cx)
first' ⟨Sum α β⟩    (Inr x)   = do (t, cx) ← first' ⟨β⟩ x
                                    return (t, Inr cx)
first' ⟨Prod α β⟩   (x₁ × x₂) = (do (t, cx₁) ← first' ⟨α⟩ x₁
                                     return (t, Inl (cx₁ × x₂)))
                                 ‘mplus‘
                                 (do (t, cx₂) ← first' ⟨β⟩ x₂
                                     return (t, Inr (x₁ × cx₂)))
firstId             x c       = Just (x, c)
first ⟨f :: ∗ → ∗⟩ ::
   (first' ⟨f⟩) ⇒ ∀(b :: ∗) (c :: ∗). f b → c → Maybe (b, Ctx ⟨f⟩ b c)
first ⟨γ :: ∗ → ∗⟩ x c        = let first' ⟨α⟩ x = firstId x c
                                 in first' ⟨γ α⟩ x
```

We pass to the *first* function the opened functor, and the context. If *first* succeeds and finds a child, it returns the child paired with the new context. We use do-notation for the Maybe monad in the definition, to simplify the presentation. The type Unit as well as primitive types of kind $*$ such as Int and Char have no children, we thus return *Nothing*. In the Sum case, we try to find the first child of whatever alternative the value we get belongs to. The Prod case is most interesting. We first try to find a child of the left component, and only if that does not exist, we proceed to look in the right component.

We can now define *down* to shift focus to the leftmost child if there is any, and to do nothing if no children exist:

```
down ⟨f :: ∗ → ∗⟩ :: (first' ⟨f⟩) ⇒ Loc ⟨f⟩ → Loc ⟨f⟩
down ⟨γ :: ∗ → ∗⟩ (x, c) = case first ⟨γ⟩ (out x) c of
                              Nothing          → (x, c)
                              Just (newx, newc) → (newx, In (Down newc)) .
```

The function *down* depends on *first'* – it is defined in terms of generic abstraction on *first*, but *first* is itself a generic abstraction over *first'*. However, *down* is intended to be used on constant types of kind $* \to *$, thus the dependency is mostly irrelevant.

16.7 Implementation of type-indexed datatypes

In this section, we describe how type-indexed datatypes can be translated. In many aspects, the algorithms needed here correspond to the mechanisms that we have introduced for type-indexed functions. And because type-indexed types are not parametrized by type tuples, but always by single types, the analogous situation for type-indexed functions that is described in Chapter 6 is the closest match.

It helps to compare the rules for type-indexed types with the counterparts for type-indexed functions, therefore the references to the corresponding rules are always provided.

Because it would lead to too much duplication of concepts, we omit some of the less interesting parts of the translation, such as the translation of environments. After having introduced the additional syntax in Section 16.7.1, we discuss several aspects of the translation: qualified kinds, generic application, types and type arguments, expressions, type declarations, and finally declarations of type-indexed functions.

16.7.1 Syntax of FCRT+gftx

Figure 16.2 shows all syntactic extensions that are necessary to cover type-indexed types, including local redefinition and generic abstraction on the type level.

We deviate from the syntax used in the examples and write type-indexed types using a **Typecase** construct (corresponding to **typecase** for type-indexed functions). Furthermore, we do not allow **type**, **newtype**, and **data** to appear directly in type-indexed type definitions, local redefinitions, and generic abstraction, but require all the right hand sides to be ordinary types. This restriction is made here to simplify the presentation.

It is, however, easy to translate the syntax used in the previous sections to the syntax that we use for the formal language. The definition for type-indexed tries, as given in Section 16.1, was

```
type FMap ⟨Int⟩        v = IntMap v
type FMap ⟨Char⟩       v = CharMap v
type FMap ⟨Unit⟩       v = Maybe v
type FMap ⟨Sum α β⟩  v = (FMap ⟨α⟩ v, FMap ⟨β⟩ v)
type FMap ⟨Prod α β⟩ v = FMap ⟨α⟩ (FMap ⟨β⟩ v) .
```

In a first step, we move all arms under a **Typecase** construct:

```
FMap ⟨a⟩ = Typecase a of
  Int      → type Λv :: ∗. IntMap v
  Char     → type Λv :: ∗. CharMap v .
```

Type declarations

D ::= ...	everything from Figure 15.1
$\mid\ T\ \langle a \rangle = \textbf{Typecase}\ a\ \textbf{of}\ \{P_i \rightarrow t_i\}_{;}^{i \in 1..n}$	type-indexed datatype declaration
$\mid\ \textbf{newtype}\ T\ \langle R \rangle\ \textbf{as}\ C$	newtype specialization request
$\mid\ \textbf{type}\ T\ \langle R \rangle$	type specialization request
$\mid\ T\ \langle \alpha :: \kappa \rangle = t$	type-level generic abstraction

Types

t, u ::= ...	everything from Figure 3.1
$\mid\ T\ \langle A \rangle$	type-level generic application
$\mid\ \textbf{Let}\ T\ \langle \alpha\ \{(\gamma_i :: \kappa_i)\}^{i \in 1..n} \rangle = t_1\ \textbf{In}\ t_2$	type-level local redefinition

Type arguments

A ::= ...	everything from Figure 6.1
$\mid\ T\ \langle A \rangle$	type-level generic application
$\mid\ \textbf{Let}\ T\ \langle \alpha\ \{(\gamma_i :: \kappa_i)\}^{i \in 1..n} \rangle = A_1\ \textbf{In}\ A_2$	type-level local redefinition

Specialization request patterns

R ::= T	named type
$\mid\ T\ \langle R \rangle$	type-indexed datatype component

Kind signatures

$\bar{\sigma}$ ::= $(\{T_k\}_{,}^{k \in 1..n}) \Rightarrow \kappa$	kind signature of type-indexed datatype

Qualified kinds

ρ ::= $(\bar{\Delta}) \Rightarrow \kappa$	qualified kind

Kind constraint sets

$\bar{\Delta}$ ::= $\{\bar{Y}_i\}_{,}^{i \in 1..n}$	kind constraint set

Kind constraints

$\bar{Y}$ ::= $T\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: \rho_0$	kind dependency constraint

Figure 16.2: Syntax of type-indexed datatypes in language FCRT+gftx

$$\begin{array}{lll}
\text{Unit} & \rightarrow & \textbf{type}\ \Lambda v :: *.\ \text{Maybe}\ v \\
\text{Sum}\ \alpha\ \beta & \rightarrow & \textbf{type}\ \Lambda v :: *.\ (\text{FMap}\ \langle\alpha\rangle\ v, \text{FMap}\ \langle\beta\rangle\ v) \\
\text{Prod}\ \alpha\ \beta & \rightarrow & \textbf{type}\ \Lambda v :: *.\ \text{FMap}\ \langle\alpha\rangle\ (\text{FMap}\ \langle\beta\rangle\ v)
\end{array}$$

Now, we assume that we have defined top-level type synonyms such that we can get rid of the **type** definitions in the arms of the **Typecase**:

$$\begin{array}{l}
\textbf{type}\ \text{FMapSum}\ \ (a :: * \rightarrow *)\ (b :: * \rightarrow *)\ (v :: *) = (a\ v, b\ v) \\
\textbf{type}\ \text{FMapProd}\ (a :: * \rightarrow *)\ (b :: * \rightarrow *)\ (v :: *) = a\ (b\ v)\ \ .
\end{array}$$

The definition of the type-indexed type FMap then becomes

$$\begin{array}{lll}
\multicolumn{3}{l}{\text{FMap}\ \langle a\rangle = \textbf{Typecase}\ a\ \textbf{of}} \\
\quad\text{Int} & \rightarrow & \text{IntMap} \\
\quad\text{Char} & \rightarrow & \text{CharMap}\ \ . \\
\quad\text{Unit} & \rightarrow & \text{Maybe} \\
\quad\text{Sum}\ \alpha\ \beta & \rightarrow & \text{FMapSum}\ \ (\text{FMap}\ \langle\alpha\rangle)\ (\text{FMap}\ \langle\beta\rangle) \\
\quad\text{Prod}\ \alpha\ \beta & \rightarrow & \text{FMapProd}\ (\text{FMap}\ \langle\alpha\rangle)\ (\text{FMap}\ \langle\beta\rangle)
\end{array}$$

This definition is legal according to the syntax in Figure 16.2. Cases declared using **data** or **newtype** constructs can be handled in a similar way. A full compiler would allow the more convenient syntax used in the examples, and implicitly introduce additional type declarations such as FMapSum and FMapProd as needed. This technique works for local redefinition and generic abstraction as well.

The other syntactic extensions are the explicit specialization requests as described in Section 16.2, generic application on the type level to instantiate type-indexed datatypes, and everything that is related to kind signatures of type-indexed datatypes and qualified kinds.

Note that generic application and local redefinition are added not only to the language of types, but also to the language of type arguments. This makes it possible to call generic functions on instances of type-indexed types, so that we can, for example, compare two type-indexed tries for equality using the generic *equal* function.

Kind signatures list as dependencies a set of named types, and have a constant *base kind*. Qualified kinds are derived internally from a kind signature using a type level generic application algorithm. They are very similar to qualified types: a kind prefixed by a list of kind dependency constraints, where each of the dependency constraints associates the name of a type-indexed type at a dependency variable (possibly with arguments) with a kind.

For some of the syntactic categories related to type-indexed datatypes (signatures, kind constraint sets, kind constraints), we reuse the latter for metavariables of the equivalent category for type-indexed functions, with a bar on top.

16.7.2 Translation of qualified kinds

$$\mathrm{K} \vdash \bar{Y}$$

$$\frac{\begin{array}{c}\mathrm{K}' \equiv \mathrm{K}\ \{,\alpha_i :: \kappa_i\}^{i \in 1..n} \\ \mathrm{K}' \vdash \alpha_0\ \{\alpha_i\}^{i \in 1..n} :: * \\ T\ \langle a :: * \rangle :: (\{T_k\}_{,}^{k \in 1..\ell}) \Rightarrow \kappa \in \mathrm{K} \\ \mathrm{K}' \vdash \rho\end{array}}{\mathrm{K} \vdash T\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: \rho} \quad \text{(k-constraint)}$$

Figure 16.3: Well-formedness of kind dependency constraints in FCRT+gftx of Figure 16.2, compare with Figure 6.4

$$\mathrm{K} \vdash \rho$$

$$\frac{\{\mathrm{K} \vdash \bar{Y}_j\}^{j \in 1..n}}{\mathrm{K} \vdash (\{\bar{Y}_j\}_{,}^{j \in 1..n}) \Rightarrow \kappa} \quad \text{(k-qual)}$$

Figure 16.4: Well-formedness of qualified kinds in FCRT+gftx of Figure 16.2, compare with Figure 6.5

Figures 16.3 and 16.4 specify under which conditions qualified kinds are well-formed: a qualified kind is well-formed if each of the dependency constraints is well-formed. A constraint is well-formed if it refers to a named type which is in scope, i.e., which has a kind signature in the kind environment, if the type argument is of kind $*$, and if the qualified kind in the constraint is well-formed.

As for type dependency constraints (cf. Section 6.1), we assume that there is a canonical order for kind dependency constraints, so that we can translate qualified kinds in a deterministic way by replacing constraints with arrows, as detailed in Figure 16.5. The idea is exactly the same as for qualified types, the situation is simpler though, because there are no quantified variables involved.

There is a subsumption relation on qualified kinds, shown in Figure 16.6. The relation is simpler than the equivalent one on qualified types because it only

$$[\![\rho_{\text{FCRT+gftx}}]\!]^{\text{gftx}} \equiv \kappa_{\text{FCRT}}$$

$$\frac{[\![\bar{\Delta}]\!]^{\text{gftx}} \equiv \kappa'[\bullet]}{[\![(\bar{\Delta}) \Rightarrow \kappa]\!]^{\text{gftx}} \equiv \kappa'[\kappa]} \quad \text{(tr-qkind)}$$

$$[\![\bar{\Delta}_{\text{FCRT+gftx}}]\!]^{\text{gftx}} \equiv \kappa_{\text{FCRT}}[\bullet]$$

$$\frac{}{[\![\varepsilon]\!]^{\text{gftx}} \equiv \bullet} \quad \text{(tr-kconstraint-1)}$$

$$\frac{\begin{array}{c}[\![\bar{\Delta}]\!]^{\text{gftx}} \equiv \kappa'[\bullet] \\ [\![\rho]\!]^{\text{gftx}} \equiv \kappa\end{array}}{[\![\bar{\Delta}, T\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: \rho]\!]^{\text{gftx}} \equiv \kappa'[\kappa \to \bullet]} \quad \text{(tr-kconstraint-2)}$$

Figure 16.5: Translation of qualified kinds and kind dependency constraints in FCRT+gftx, compare with Figure 6.9

$$\bar{\Delta} \vdash \rho_1 \leqslant \rho_2 \rightsquigarrow t[\bullet]$$

$$\frac{\bar{\Delta}_1, \bar{\Delta}_2 \vdash \rho \leqslant \kappa \rightsquigarrow t[\bullet] \qquad \vdash \bar{\Delta}_2 \rightsquigarrow t_{\bar{\Delta}}[\bullet]}{\bar{\Delta}_1 \vdash \rho \leqslant (\bar{\Delta}_2) \Rightarrow \kappa \rightsquigarrow t_{\bar{\Delta}}[t[\bullet]]} \quad \text{(rhos-dep-1)}$$

$$\frac{\bar{\Delta}_2 \Vdash \bar{\Delta}_1 \rightsquigarrow t[\bullet]}{\bar{\Delta}_2 \vdash (\bar{\Delta}_1) \Rightarrow \kappa \leqslant \kappa \rightsquigarrow t[\bullet]} \quad \text{(rhos-dep-2)}$$

Figure 16.6: Subsumption relation on qualified kinds, compare with Figure 6.10

involves the addition or removal of kind dependency constraints, whereas the type-level equivalent (cf. Figure 6.10) must handle universal quantifications as well. The rules in Figure 16.6 make use of a conversion and an entailment judgment, of the forms

$$\begin{array}{l} \vdash \bar{\Delta} \rightsquigarrow t[\bullet] \\ \bar{\Delta}_1 \Vdash \bar{\Delta}_2 \rightsquigarrow t[\bullet] \ . \end{array}$$

They correspond directly to the judgments displayed in Figures 6.11 and 6.12 on pages 87 and 88, and do not provide anything new. They are therefore omitted here.

16.7.3 Translation of generic application

A central part of the treatment of type-indexed datatypes is the translation of generic application on the type level. We present two algorithms: the generic application algorithm Gapp, which corresponds to gapp on pages 78 and 100 in Figures 6.8 and 6.19; and the generic translation algorithm $[\![\cdot]\!]^{\mathrm{Gtrans}}$, which corresponds to $[\![\cdot]\!]^{\mathrm{gtrans}}$ on page 94 in Figure 6.18. The former assigns a qualified kind to a generic application, using the kind signature of the type-indexed datatype, while the latter specializes a generic application to an application of components of a type-indexed datatype.

Before we can proceed to the first of the two, we require some judgments to access information on the kind signatures. Kind signatures are stored in the kind environment, as we have already assumed in the rules for the well-formedness of qualified kinds. Kind signatures have a **base kind**, which is the part to the right of the list of **dependencies** and the double arrow. Other than the base type, the base kind of a type-indexed datatype is constant. The base kind of a type-indexed datatype T can be accessed using a judgment of the form

$$\mathsf{Base}_{\mathrm{K}}(T) \equiv \kappa \ .$$

The dependencies can be determined using a judgment of the form

$$\mathsf{Dependencies}_{\mathrm{K}}(T) \equiv \{T_k\}_{,}^{k \in 1..n} \ .$$

The corresponding rules are presented in Figure 16.7.

In Figure 16.8, we show how to check the well-formedness of a kind signature: a transitivity condition must hold such as in rule (typesig) in Figure 6.7 for type-indexed functions, and for reasons analogous to those explained in Section 5.4; all type-indexed datatypes named in the dependencies must themselves have valid kind signatures, and all indirect dependencies must also be direct dependencies.

$$\mathsf{Base}_{\mathrm{K}}(T) \equiv \kappa$$

$$\frac{T\ \langle a :: *\rangle :: (\{T_k\}_{,}^{k\in 1..n}) \Rightarrow \kappa \in \mathrm{K}}{\mathsf{Base}_{\mathrm{K}}(T) \equiv \kappa} \quad \text{(Base)}$$

$$\mathsf{Dependencies}_{\mathrm{K}}(T) \equiv \{T_k\}_{,}^{k\in 1..n}$$

$$\frac{T\ \langle a :: *\rangle :: (\{T_k\}_{,}^{k\in 1..n}) \Rightarrow \kappa \in \mathrm{K}}{\mathsf{Dependencies}_{\mathrm{K}}(T) \equiv \{T_k\}_{,}^{k\in 1..n}} \quad \text{(Deps)}$$

Figure 16.7: Extracting information from the kind signature of a type-indexed datatype, compare with Figure 6.6

$$\mathrm{K} \vdash^{ksig} T\ \langle a :: *\rangle :: \bar{\sigma}$$

$$\frac{\begin{array}{c}\left\{\mathrm{K} \vdash^{ksig} T_k\ \langle a :: *\rangle :: (\{T_{k,i}\}_{,}^{i\in 1..m_k}) \Rightarrow \kappa_k\right\}^{k\in 1..n} \\ \left\{\{T_{k,i} \in \{T_j\}_{,}^{j\in 1..n}\}^{i\in 1..m_k}\right\}^{k\in 1..n}\end{array}}{\mathrm{K} \vdash^{ksig} T\ \langle a :: *\rangle :: (\{T_k\}_{,}^{k\in 1..n}) \Rightarrow \kappa} \quad \text{(kindsig)}$$

Figure 16.8: Well-formedness of kind signatures for type-indexed datatypes, compare with Figure 6.7

$$\mathsf{Gapp}_{\mathrm{K}}(T\ \langle A\rangle) \equiv \rho$$

$$\frac{\mathrm{K} \vdash A :: * \qquad \mathsf{fdv}(A) \equiv \varepsilon}{\mathsf{Gapp}_{\mathrm{K}}(T\ \langle A\rangle) \equiv \mathsf{Base}_{\mathrm{K}}(T)} \quad \text{(Ga-1)}$$

$$\frac{\mathrm{K} \vdash \alpha :: \kappa \qquad a \text{ fresh} \qquad \mathrm{K}' \equiv \mathrm{K}, a :: \kappa}{\begin{array}{l}\mathsf{Gapp}_{\mathrm{K}}(T\ \langle \alpha\ \{A_i\}^{i\in 1..n}\rangle) \equiv (\mathsf{Mkdep}_{\mathrm{K}'}(T\ \langle \alpha \leftarrow a\rangle)) \\ \qquad\qquad\qquad\qquad \Rightarrow \mathsf{Gapp}_{\mathrm{K}'}(T\ \langle a\ \{A_i\}^{i\in 1..n}\rangle)\end{array}} \quad \text{(Ga-2)}$$

$$\frac{\begin{array}{c}\alpha \in \mathsf{fdv}(A) \qquad \mathsf{head}(A) \not\equiv \alpha \\ \mathrm{K} \vdash \alpha :: \kappa \qquad a \text{ fresh} \qquad \mathrm{K}' \equiv \mathrm{K}, a :: \kappa \\ \mathsf{Dependencies}_{\mathrm{K}}(T) \equiv \{T_k\}_{,}^{k\in 1..n}\end{array}}{\begin{array}{l}\mathsf{Gapp}_{\mathrm{K}}(T\ \langle A\rangle) \equiv (\{\mathsf{Mkdep}_{\mathrm{K}'}(T_k\ \langle \alpha \leftarrow a\rangle)\}_{,}^{k\in 1..n}) \\ \qquad\qquad\qquad\qquad \Rightarrow \mathsf{Gapp}_{\mathrm{K}'}(T\ \langle A[a\ /\ \alpha]\rangle)\end{array}} \quad \text{(Ga-3)}$$

$$\frac{\begin{array}{c}\mathsf{fdv}(A) \equiv \varepsilon \qquad \mathrm{K} \vdash A :: \kappa \to \kappa' \\ a \text{ fresh} \qquad \mathrm{K}' \equiv \mathrm{K}, a :: \kappa \\ \mathsf{Dependencies}_{\mathrm{K}}(T) \equiv \{T_k\}_{,}^{k\in 1..n}\end{array}}{\mathsf{Gapp}_{\mathrm{K}}(T\ \langle A\rangle) \equiv \{\mathsf{Gapp}_{\mathrm{K}'}(T_k\ \langle a\rangle) \to\}^{k\in 1..n}\ \mathsf{Gapp}_{\mathrm{K}'}(T\ \langle A\ a\rangle)} \quad \text{(Ga-4)}$$

$$\mathsf{Mkdep}_{\mathrm{K}}(T\ \langle \alpha \leftarrow a\rangle) \equiv \bar{Y}$$

$$\frac{\begin{array}{c}\mathrm{K} \vdash \alpha :: \kappa \qquad \kappa \equiv \{\kappa_h \to\}^{h\in 1..\ell}\ * \\ \{\gamma_h \text{ fresh}\}^{h\in 1..\ell} \qquad \mathrm{K}' \equiv \mathrm{K}\ \{, \gamma_h :: \kappa_h\}^{h\in 1..\ell}\end{array}}{\begin{array}{l}\mathsf{Mkdep}_{\mathrm{K}}(T\ \langle \alpha \leftarrow a\rangle) \\ \qquad \equiv T\ \langle \alpha\ \{\gamma_h\}^{h\in 1..\ell}\rangle :: \mathsf{Gapp}_{\mathrm{K}}(T\ \langle a\ \{\gamma_h\}^{h\in 1..\ell}\rangle)\end{array}} \quad \text{(Mkdep)}$$

Figure 16.9: Generic application algorithm for type-indexed datatypes, compare with Figures 6.8 and 6.19

$$[\![T\ \langle A\rangle]\!]^{\mathrm{Gtrans}}_{\mathrm{K};\bar{\Sigma};\Psi} \equiv t_{\mathrm{FCRT}}$$

$$\frac{T\ \langle T'\rangle \in \bar{\Sigma}}{[\![T\ \langle T'\rangle]\!]^{\mathrm{Gtrans}}_{\mathrm{K};\bar{\Sigma};\Psi} \equiv \mathsf{Cp}(T, T')} \quad \text{(Gtr-named)}$$

$$\frac{T\ \langle a\rangle \in \bar{\Sigma}}{[\![T\ \langle a\rangle]\!]^{\mathrm{Gtrans}}_{\mathrm{K};\bar{\Sigma};\Psi} \equiv \mathsf{Cp}(T, a)} \quad \text{(Gtr-var)}$$

$$\frac{\mathsf{Cp}(T, \alpha) \equiv t \in \Psi}{[\![T\ \langle \alpha\rangle]\!]^{\mathrm{Gtrans}}_{\mathrm{K};\bar{\Sigma};\Psi} \equiv t} \quad \text{(Gtr-depvar)}$$

$$\frac{\mathsf{Dependencies}_{\mathrm{K}}(T) \equiv \{T_k\}^{k \in 1..\ell}_{,}}{[\![T\ \langle A_1\ A_2\rangle]\!]^{\mathrm{Gtrans}}_{\mathrm{K};\bar{\Sigma};\Psi} \equiv [\![T\ \langle A_1\rangle]\!]^{\mathrm{Gtrans}}_{\mathrm{K};\bar{\Sigma};\Psi}\ \{[\![T_k\ \langle A_2\rangle]\!]^{\mathrm{Gtrans}}_{\mathrm{K};\bar{\Sigma};\Psi}\}^{k \in 1..\ell}} \quad \text{(Gtr-app)}$$

Figure 16.10: Translation of generic application of type-indexed datatypes in language FCR+tif+par, compare with Figure 6.18

Now let us have a look at the generic application algorithm on the type level, in Figure 16.9. The four rules of the algorithm Gapp and the auxiliary judgment Mkdep to create a single dependency constraint, correspond directly to the rules in Figures 6.8 and 6.19 for gapp and mkdep. Rule (Ga-1) returns the base kind if the type argument is without dependency variables and of kind $*$. For type arguments with dependency variables, kind dependencies are introduced in rules (Ga-2) and (Ga-3), using the rule (Mkdep) to generate the actual constraint. Rule (Ga-2) handles the special case that a dependency variable α is the head of the type argument of type-indexed type T, which always causes a dependency on T for α. The rule (Ga-4) is for type arguments without dependency variables, but of a kind different than $*$. This rule is similar to (Ga-3), but instead of dependency constraints, (type-level) function arguments are introduced.

Using this algorithm, one can determine the kinds that instances of type-indexed types must have for specific type arguments. For example, we can verify the kinds given in Figure 16.1 on page 238 for applications of FMap to different type arguments.

The associated translation algorithm in Figure 16.10 has judgments of the form

$$[\![T \ \langle A \rangle]\!]^{\text{Gtrans}}_{\text{K};\Sigma;\Psi} \equiv t_{\text{FCRT}} ,$$

transforming a generic application of a type-indexed datatype T to a type argument A into an FCRT type t_{FCRT}. If the type argument of the application is a named type T', then rule (Gtr-named) replaces the application with a reference to the appropriate component, under the condition that an entry $T \ \langle T' \rangle$ is contained in the signature environment. Recall that we use Cp to produce the names of components for type-indexed datatypes.

A type variable is treated in the same way as a named type, as described by rule (Gtr-var). Generic application to normal (not dependency) type variables occurs sometimes for universally quantified type variables. The translation of universal quantification is such that additional type variables for components of the form $\mathsf{Cp}(T, a)$ may be brought into scope. This translation of universal quantification is described in Section 16.7.4.

If the type argument is a dependency variable, as in rule (Gtr-depvar), we check environment Ψ for an entry of the form $\mathsf{Cp}(T, \alpha) \equiv t$, and use t as translation. To keep the rule simple, we implicitly assume that Ψ contains entries of the form $\mathsf{Cp}(T, \alpha) \equiv \mathsf{Cp}(T, \alpha)$ for each type-indexed type T and each dependency variable α in scope. These default entries are overwritten if a type-level local redefinition is encountered, as described in Section 16.7.4. Because the target language FCRT provides no type-level **Let**, we choose to store local redefinitions in the environment Ψ and apply them here, during the translation of generic application.

The final rule, (Gtr-app), translates a generic application of T to a type argument that consists itself of an application $(A_1 \ A_2)$. The result is the translation of $T \ \langle A_1 \rangle$, applied to all $T_k \ \langle A_2 \rangle$, where the T_k are the dependencies of T. As in the algorithm for type-indexed functions, in Figure 6.18 on page 94, we thus translate an application in the type argument to an application of the translations, and keep the whole algorithm compositional. We have described in Section 6.6 that the way in which we translate generic applications of type-indexed functions corresponds to the use of kind-indexed types internally. In analogy, our translation here treats type-indexed types as if they possessed kind-indexed kinds. This analogy is described in more detail in the paper on type-indexed datatypes (Hinze *et al.* 2002).

The translation algorithm just described cannot handle type arguments containing generic application or local redefinition. We will see in the next section that we simplify these constructs prior to calling the algorithm.

$$[\![t_{\text{FCRT+gftx}} :: \kappa_{\text{FCRT+gftx}}]\!]^{\text{gftx}}_{\mathrm{K};\bar{\Delta};\bar{\Sigma};\Psi} \equiv t_{\text{FCRT}}$$

$$\frac{\begin{array}{c} T\ \langle a :: *\rangle :: \bar{\sigma} \in \mathrm{K} \\ \mathrm{K}' \equiv \mathrm{K}\ \{, \gamma_i :: \kappa_i\}^{i\in 1..n} \qquad \kappa' \equiv \{\kappa_i \to\}^{i\in 1..n} * \\ \Psi' \equiv \Psi, \mathsf{Cp}(T,\alpha) = t_1' \\ \bar{\Delta}' \equiv \bar{\Delta}, T\ \langle \alpha\ \{(\gamma_i :: \kappa_i)\}^{i\in 1..n}\rangle :: \rho_1 \\ [\![t_1 :: \rho_1]\!]^{\text{gftx}}_{\mathrm{K}';\bar{\Delta};\bar{\Sigma};\Psi} \equiv t_1' \qquad [\![t_2 :: \kappa]\!]^{\text{gftx}}_{\mathrm{K},\alpha::\kappa';\bar{\Delta}';\bar{\Sigma};\Psi'} \equiv t_2' \end{array}}{[\![\mathbf{Let}\ T\ \langle \alpha\ \{(\gamma_i :: \kappa_i)\}^{i\in 1..n}\rangle = t_1\ \mathbf{In}\ t_2 :: \kappa]\!]^{\text{gftx}}_{\mathrm{K};\bar{\Delta};\bar{\Sigma};\Psi} \equiv t_2'} \quad \text{(t/tr-let-lr)}$$

$$\frac{\begin{array}{c} \{\mathsf{Base}_{\mathrm{K}}(T_i) \equiv \kappa_i\}^{i\in 1..n} \\ \mathrm{K}' \equiv \mathrm{K}, a :: \kappa \qquad \Sigma' \equiv \Sigma\ \{, T_i\ \langle a\rangle\}^{i\in 1..n} \\ [\![t :: *]\!]^{\text{gftx}}_{\mathrm{K}';\bar{\Delta};\bar{\Sigma}';\Psi} \equiv t' \end{array}}{[\![\forall a :: \kappa.\ t :: *]\!]^{\text{gftx}}_{\mathrm{K};\bar{\Delta};\bar{\Sigma};\Psi} \equiv \forall a :: \kappa.\ \{\forall(\mathsf{Cp}(T_i, a) :: \kappa_i).\}^{i\in 1..n}\ t'} \quad \text{(t/tr-forall)}$$

$$\frac{\begin{array}{c} [\![A :: *]\!]^{\text{gftx}}_{\mathrm{K};\bar{\Delta};\bar{\Sigma};\Psi} \equiv A' \\ \mathsf{Gapp}_{\mathrm{K}}(T\ \langle A'\rangle) \equiv \rho \qquad [\![T\ \langle A'\rangle]\!]^{\text{Gtrans}}_{\mathrm{K};\bar{\Sigma};\Psi} \equiv t \\ \mathrm{K}; \bar{\Delta} \vdash \rho \leqslant \kappa \end{array}}{[\![T\ \langle A\rangle :: \kappa]\!]^{\text{gftx}}_{\mathrm{K};\bar{\Delta};\bar{\Sigma};\Psi} \equiv t} \quad \text{(t/tr-genapp)}$$

Figure 16.11: Translation of FCRT+gftx types to FCRT

$$[\![t_{\text{FCRT}+\text{gftx}} :: \rho_{\text{FCRT}+\text{gftx}}]\!]^{\text{gftx}}_{\mathrm{K};\bar{\Delta};\bar{\Sigma};\Psi} \equiv t_{\text{FCRT}}$$

$$\frac{\begin{array}{c}[\![t :: \kappa]\!]^{\text{gftx}}_{\bar{\Delta},\bar{\Delta}'} \equiv t' \\ \bar{\Delta} \vdash \kappa \leqslant (\bar{\Delta}') \Rightarrow \kappa \rightsquigarrow t''[\bullet]\end{array}}{[\![t :: (\bar{\Delta}') \Rightarrow \kappa]\!]^{\text{gftx}}_{\bar{\Delta}} \equiv t''[t']} \quad \text{(t/tr-reveal)}$$

$$\frac{\begin{array}{c}[\![t :: \rho_1]\!]^{\text{gftx}} \equiv t' \\ \vdash \rho_1 \leqslant \rho_2 \rightsquigarrow t''[\bullet]\end{array}}{[\![t :: \rho_2]\!]^{\text{gftx}} \equiv t''[t']} \quad \text{(t/tr-subs)}$$

Figure 16.12: Revelation of kind dependency constraints in FCRT+gftx, compare with Figure 6.14

16.7.4 Translation of types and type arguments

With the rules for handling generic application available, we can now tackle kind checking and translation of types and type arguments. Judgments are of the form

$$[\![t_{\text{FCRT}+\text{gftx}} :: \kappa_{\text{FCRT}+\text{gftx}}]\!]^{\text{gftx}}_{\mathrm{K};\bar{\Delta};\bar{\Sigma};\Psi} \equiv t_{\text{FCRT}} \;,$$

expressing that type $t_{\text{FCRT}+\text{gftx}}$ of kind $\kappa_{\text{FCRT}+\text{gftx}}$ translates to type t_{FCRT}. We need the kind environment K, containing – among other things – the kind signatures of type-indexed datatypes, a kind dependency environment $\bar{\Delta}$, a signature environment $\bar{\Sigma}$ for type-indexed datatypes, and the environment Ψ for type synonyms. The rules are given in Figure 16.11.

Next to the two new constructs in the type language, local redefinition and generic application, also the translation of universal quantification needs discussion. Rule (t/tr-let-lr) is for the type-level **Let** construct, which exists only for the purpose of local redefinition. In contrast to the expression-level **let**, it contains only one binding and is non-recursive. In the examples, we have used the construct with multiple bindings, which is to be understood as syntactic sugar for a nested **Let** statement. The type-indexed type T that is redefined must be in scope, i.e., its kind signature must be in the kind environment K. We translate the right hand side of the redefinition, t_1, to t_1', under an environment K' which contains the argument dependency variables γ_i. Note that t_1 may be of a qualified kind ρ_1. To allow this, we use the modified type checking rules in Figure 16.12 that can be used to reveal the kind dependencies of a type. The body of the let construct, t_2, is translated under an environment that contains the dependency variable α for which the local redefinition occurs. Furthermore, it is passed extended environments $\bar{\Delta}'$, containing the redefinition, and Ψ', which contains a mapping from $\mathsf{Cp}(T, \alpha)$ to t_1'. This entry can be used to translate generic applications that occur in the body (cf. Figure 16.10).

In many cases, a universal quantification can be translated to itself. However, if the body refers to a type-indexed datatype, say T, and the quantified variable a occurs in its type argument, then the translation must contain additional universal quantifications. These additional quantified variables take the form of a component, $\mathsf{Cp}(T, a)$. We then also extend the type-level signature environment with an entry $T \ \langle a \rangle$ for the scope of the quantifier, such that they are available while translating the generic application (again, cf. Figure 16.10). There is an example below.

The rule (t/tr-genapp) translates a generic application. The type argument A is recursively translated to A' first, thereby removing possible further generic applications, quantifications, or local redefinitions. Therefore, A' is in a form such that we can find the kind of the application by an invocation of Gapp. The dependency constraints of that kind must be satisfiable in the current type-level dependency environment $\bar{\Delta}$. We can then use $[\![\cdot]\!]^{\text{gtrans}}$, also on A', to determine the final translation.

As an example application of the translation rules for types, consider the function *lookup* (defined in Section 16.1), which is of type

$$lookup \ \langle a :: * \rangle :: (lookup \ \langle a \rangle) \Rightarrow \forall v :: *.\ a \to \text{FMap} \ \langle a \rangle \ v \to \text{Maybe} \ v \ .$$

A call *lookup* $\langle [\alpha] \rangle$ is of type

$$\begin{aligned} &\forall a :: *.\ (lookup \ \langle \alpha \rangle :: \forall v :: *.\ a \to \text{FMap} \ \langle a \rangle \ v \to \text{Maybe} \ v) \\ &\qquad\qquad \Rightarrow \forall v :: *.\ [a] \to \text{FMap} \ \langle [a] \rangle \ v \to \text{Maybe} \ v \ . \end{aligned}$$

According to the rules given in this section, the corresponding FCRT translation of this type is

$$\begin{aligned} &\forall (a :: *) \ (\mathsf{Cp}(\text{FMap}, a) :: * \to *). \\ &\quad (\forall v :: *.\ a \to \mathsf{Cp}(\text{FMap}, a) \ v \to \text{Maybe} \ v) \\ &\qquad \to \forall v :: *.\ [a] \to \mathsf{Cp}(\text{FMap}, [\,]) \ \mathsf{Cp}(\text{FMap}, a) \ v \to \text{Maybe} \ v \ . \end{aligned}$$

The rules of Figure 16.11 can be applied also to type arguments, and can be extended canonically to qualified types.

$$[\![e_1 :: t_{\text{FCRT+gftx}}]\!]^{\text{gftx}}_{K;\Gamma;\Sigma;\bar{\Sigma};\Psi} \equiv e_2$$

$$\frac{[\![t :: *]\!]^{\text{gftx}}_{K;\varepsilon;\bar{\Sigma};\Psi} \equiv t' \qquad [\![e :: t']\!]^{\text{gftx}} \equiv e'}{[\![e :: t]\!]^{\text{gftx}}_{K;\Gamma;\Sigma;\bar{\Sigma};\Psi} \equiv e'} \quad \text{(e/tr-gtype)}$$

Figure 16.13: Translation of FCRT+gftx expressions to FCRT, extends Figures 6.13, 8.1, and 12.3

16.7.5 Translation of expressions

Expressions can now have types involving the new constructs, such as generic application or local redefinition. We therefore add an additional rule (e/tr-gtype) to the checking and translation rules for expressions, that states that an expression e can be of type t if it can also be of type t', where t' is the translation of t according to the rules stated in Section 16.7.4. The additional rule, which extends the previous rules for expressions, is displayed in Figure 16.13.

16.7.6 Translation of type declarations

The type declaration language is significantly richer in the presence of type-indexed datatypes. Therefore, it is worthwhile to analyze how type declarations are checked and translated. We present selected rules in the Figures 16.14 and 16.15. Judgments are of the form

$$[\![D_{\text{FCRT+gftx}} \rightsquigarrow K_2; \Gamma; \bar{\Sigma}_2; \Psi_2; E]\!]^{\text{gftx}}_{K_1;\bar{\Sigma}_1;\Psi_1} \equiv \{D_{\text{FCRT}}\}^{i \in 1..n}_{;} \ .$$

Under a kind environment K_1, a type-level signature environment $\bar{\Sigma}_1$, and a type synonym environment Ψ_1, the type declaration $D_{\text{FCRT+gftx}}$ is translated into a sequence of FCRT type declarations $\{D_{\text{FCRT}}\}^{i \in 1..n}_{;}$. As a side product, we get environments that contain new information won from the declaration: kind information in K_2, constructor information in Γ, components of type-indexed datatypes in $\bar{\Sigma}_2$, type synonyms including structural representation types in Ψ_2, and embedding-projection pairs in E.

Rule (p/tr-tid) explains how to translate a type-indexed datatype, and closely corresponds to rule (d/tr-tif) on page 91 for type-indexed functions. We delegate the translation of each of the arms to an auxiliary rule (p/tr-arm), which transforms the arm into a type synonym. For the whole **Typecase** construct, we then

$$\llbracket D_{\textsc{fcrt}+\text{gftx}} \rightsquigarrow \mathrm{K}_2;\Gamma;\bar{\Sigma}_2;\Psi_2;\mathrm{E} \rrbracket^{\text{gftx}}_{\mathrm{K}_1;\bar{\Sigma}_1;\Psi_1} \equiv \{D_{\textsc{fcrt}}\}^{i\in 1..n}_{;}$$

$$\frac{\begin{array}{c} \mathrm{K} \vdash^{ksig} T\ \langle a :: *\rangle :: \bar{\sigma} \\ \mathrm{K}' \equiv T\ \langle a :: *\rangle :: \bar{\sigma}\ \{,\mathrm{K}_i\}^{i\in 1..n} \qquad \bar{\Sigma}' \equiv \{\bar{\Sigma}_i\}^{i\in 1..n}_{,} \\ \Gamma' \equiv \{\Gamma_i\}^{i\in 1..n}_{,} \qquad \Psi' \equiv \{\Psi_i\}^{i\in 1..n}_{,} \qquad \mathrm{E}' \equiv \{\mathrm{E}_i\}^{i\in 1..n}_{,} \\ \left\{\llbracket T \mid P_i \to t_i \rightsquigarrow \mathrm{K}_i;\Gamma_i;\bar{\Sigma}_i;\Psi_i;\mathrm{E}_i \rrbracket^{\text{gftx}}_{\mathrm{K};\bar{\Sigma};\Psi} \equiv D_i\right\}^{i\in 1..n} \end{array}}{\begin{array}{r} \llbracket T\ \langle a\rangle = \textbf{Typecase}\ a\ \textbf{of}\ \{P_i \to t_i\}^{i\in 1..n}_{;} \rightsquigarrow \mathrm{K}';\Gamma';\bar{\Sigma}';\Psi';\mathrm{E}' \rrbracket^{\text{gftx}}_{\mathrm{K};\bar{\Sigma};\Psi} \\ \equiv \{D_i\}^{i\in 1..n}_{;} \end{array}} \quad \text{(p/tr-tid)}$$

$$\llbracket T \mid P \to t_{\textsc{fcrt}+\text{gftx}} \rightsquigarrow \mathrm{K}_2;\Gamma;\bar{\Sigma}_2;\Psi_2;\mathrm{E} \rrbracket^{\text{gftx}}_{\mathrm{K}_1;\bar{\Sigma}_1;\Psi_1} \equiv D_{\textsc{fcrt}}$$

$$\frac{\begin{array}{c} \mathrm{K} \vdash^{pat} P :: * \rightsquigarrow \mathrm{K}_P \qquad P \equiv T'\ \{\alpha_i\}^{i\in 1..n} \\ \mathsf{Gapp}_{\mathrm{K},\mathrm{K}_P}(T\ \langle P\rangle) \equiv \rho \qquad \llbracket t :: \rho \rrbracket^{\text{gftx}}_{\mathrm{K},\mathrm{K}_P;\varepsilon;\bar{\Sigma};\Psi} \equiv \{\Lambda a_k :: \kappa_k.\}^{k\in 1..\ell}\ t' \\ \llbracket \textbf{type}\ \mathsf{Cp}(T,T') = \{\Lambda a_k :: \kappa_k.\}^{k\in 1..\ell}\ t' \rightsquigarrow \mathrm{K}';\Gamma';\varepsilon;\Psi';\mathrm{E}' \rrbracket^{\text{gftx}}_{\mathrm{K};\bar{\Sigma};\Psi} \equiv D \end{array}}{\llbracket T \mid P \to t \rightsquigarrow \mathrm{K}';\Gamma';T\ \langle T'\rangle;\Psi';\mathrm{E}' \rrbracket^{\text{gftx}}_{\mathrm{K};\bar{\Sigma};\Psi} \equiv D} \quad \text{(p/tr-arm)}$$

Figure 16.14: Translation of FCRT+gftx type declarations to FCRT, compare with Figure 6.15

$$[\![D_{\text{FCRT+gftx}} \rightsquigarrow \mathrm{K}_2; \Gamma; \bar{\Sigma}_2; \Psi_2; \mathrm{E}]\!]^{\text{gftx}}_{\mathrm{K}_1;\bar{\Sigma}_1;\Psi_1} \equiv \{D_{\text{FCRT}}\}^{i\in 1..n}_{;}$$

$$\frac{\begin{array}{c} D \equiv \textbf{data}\ T = \{\Lambda a_i :: \kappa_i.\}^{i\in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}^{j\in 1..m}_{|} \\ \mathrm{K}_a \equiv \mathrm{K}\ \{, a_i :: \kappa_i\}^{i\in 1..\ell} \qquad \mathrm{K}' \equiv T :: \{\kappa_i \to\}^{i\in 1..\ell}\ \kappa_0 \\ \Big\{\{[\![t_{j,k} :: *]\!]^{\text{gftx}}_{\mathrm{K}_a;\varepsilon;\bar{\Sigma};\Psi} \equiv t'_{j,k}\}^{k\in 1..n_j}\Big\}^{j\in 1..m} \\ D' \equiv \textbf{data}\ T = \{\Lambda a_i :: \kappa_i.\}^{i\in 1..\ell}\ \{C_j\ \{t'_{j,k}\}^{k\in 1..n_j}\}^{j\in 1..m}_{|} \\ \Gamma' \equiv \{C_j :: \{\forall a_i :: \kappa_i.\}^{i\in 1..\ell}\ \{t_{j,k} \to\}^{k\in 1..n_j}\ T\ \{a_i\}^{i\in 1..\ell}\}^{j\in 1..m}_{,} \end{array}}{[\![D \rightsquigarrow \mathrm{K}'; \Gamma'; \varepsilon; [\![D']\!]^{\text{str}}; [\![D']\!]^{\text{ep}}]\!]^{\text{gftx}}_{\mathrm{K};\bar{\Sigma};\Psi} \equiv D'} \quad \text{(p/tr-data)}$$

$$\frac{\begin{array}{c} \mathsf{Join}(R) \equiv T' \qquad \{\alpha_k\}^{k\in 1..\ell}\ \text{fresh} \\ P \equiv T'\ \{\alpha_k\}^{k\in 1..\ell} \qquad \mathrm{K} \vdash^{pat} P :: * \rightsquigarrow \mathrm{K}_P \\ \mathsf{Str}(T') \equiv \{\Lambda a_k :: \kappa_k.\}^{k\in 1..\ell}\ t \in \Psi \\ \mathsf{Gapp}_{\mathrm{K},\mathrm{K}_P}(T\ \langle P\rangle) \equiv \rho \\ [\![T\ \langle t\{[\alpha_k\ /\ a_k]\}^{k\in 1..\ell}\rangle :: \rho]\!]^{\text{gftx}}_{\mathrm{K},\mathrm{K}_P;\varepsilon;\bar{\Sigma}} \equiv \{\Lambda b_i :: \kappa'_i.\}^{i\in 1..n}\ t' \\ D \equiv \textbf{newtype}\ \mathsf{Cp}(T, T') = \{\Lambda b_i :: \kappa'_i.\}^{i\in 1..n}\ C\ t' \\ [\![D \rightsquigarrow \mathrm{K}'; \Gamma'; \varepsilon; \Psi'; \mathrm{E}']\!]^{\text{gftx}}_{\mathrm{K};\bar{\Sigma};\Psi} \equiv D' \end{array}}{[\![\textbf{newtype}\ T\ \langle R\rangle\ \textbf{as}\ C \rightsquigarrow \mathrm{K}'; \Gamma'; T\ \langle T'\rangle; \Psi'; \mathrm{E}']\!]^{\text{gftx}}_{\mathrm{K};\bar{\Sigma};\Psi} \equiv D'} \quad \text{(p/tr-sr-newtype)}$$

Figure 16.15: Translation of FCRT+gftx declarations to FCRT, continued from Figure 16.14

only add the kind signature, and collect all the environments that result from processing the arms.

For each arm, we check that the type pattern is well-formed. We determine the qualified kind ρ that the right hand side t must have using the Gapp algorithm, and then translate t using ρ into a type expression $\{\Lambda a_k :: \kappa_k.\}^{k \in 1..\ell}\ t'$. This resulting type forms the right hand side of the type synonym $\mathsf{Cp}(T, T')$, and will in general be parametrized, because ρ can have dependency constraints. The whole type synonym is then translated again! This means that the generated type synonym is treated in the same way as a user-defined type synonym. A structure type and an embedding-projection pair are generated, and the kind information is added to the kind environment. This is necessary to allow nested generic applications such as $T''\ \langle T\ \langle P \rangle\rangle$ (cf. Section 16.8). Returned are the declaration and the environments resulting from the translation of the type synonym for $\mathsf{Cp}(T, T')$. In addition, the entry $T\ \langle T' \rangle$ is added to the signature environment.

Rule (p/tr-data) is for the translation of a datatype. Types declared by **newtype** or **type** constructs have similar rules – they are omitted here. All fields $t_{j,k}$ of all constructors are checked to be of kind $*$ and translated into fields $t'_{j,k}$, which make up the resulting FCRT datatype declaration D'. The constructors are added to the type environment, and the embedding-projection pair as well as the structural representation are generated. The generation of those is based on the translated type D'; the algorithms from Chapter 10 are therefore still valid.

The rule (p/tr-sr-newtype) translates a **newtype** specialization request. The request is of the form $T\ \langle R \rangle$, where R – according to the syntax in Figure 16.2 can be either a named type (this is the common case), or a type application. Each request is translated into a single component. To determine the name of the request, we make use of the function Join, which collapses a possibly nested type application into a single name, and is defined as follows:

$$\begin{aligned}
&\mathsf{Join}(T) && \equiv T \\
&\mathsf{Join}(T\ \langle R \rangle) && \equiv \mathsf{Cp}(T, \mathsf{Join}(R))\ .
\end{aligned}$$

In the rule, we assign the name T' to $\mathsf{Join}(R)$. This joined type T' is assumed to be already in scope, i.e., in K, and to have a structural representation in Ψ. We therefore create fresh dependency variables α_k as arguments for T', such that the pattern P, defined as $T'\ \{\alpha_k\}^{k \in 1..\ell}$ is of kind $*$. We then translate T, applied to the structural representation of T', where $\mathsf{Gapp}(T\ \langle P \rangle)$ gives us the kind of that generic application. The resulting type expression contains abstractions that correspond to the type-level dependencies. This type, including the abstractions, forms the body of the **newtype** component D that we generate under the name $\mathsf{Cp}(T, T')$, with the user-specified constructor plugged in. The declaration D is then translated again, as if it would be a normal **newtype** occurring in the program. This adds the component, together with its structural representation and embedding-projection pair, to the appropriate environments. In the final result, we also add $T\ \langle T' \rangle$ to the signature environment.

Let us play this rule through for the specialization request

$$\textbf{newtype}\ \text{FMap}\ \langle [\,] \rangle\ \textbf{as}\ \mathit{FMapList}\ .$$

Here, $R \equiv [\,]$ and $\mathsf{Join}(R) \equiv [\,]$. Because the kind of $[\,]$ is $* \to *$, we have $\ell \equiv 1$ and introduce one dependency variable $\alpha \equiv \alpha_1$. The pattern P is $[\alpha]$, and

$$\mathsf{gapp}(\text{FMap}\ \langle [\alpha] \rangle) \equiv (\text{FMap}\ \langle \alpha \rangle :: * \to *) \Rightarrow * \to * \equiv \rho\ .$$

We now translate FMap applied to the structural representation of lists:

$$\begin{aligned}
&[\![\text{FMap}\ \langle \text{Sum Unit}\ (\text{Prod}\ \alpha\ [\alpha]) \rangle :: \rho]\!]^{\text{gftx}} \\
&\quad \equiv \Lambda(\mathsf{Cp}(\text{FMap}, \alpha) :: * \to *). \\
&\qquad\qquad \mathsf{Cp}(\text{FMap}, \text{Sum})\ \mathsf{Cp}(\text{FMap}, \text{Unit}) \\
&\qquad\qquad\qquad\qquad (\mathsf{Cp}(\text{FMap}, \text{Prod})\ \mathsf{Cp}(\text{FMap}, \alpha) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{Cp}(\text{FMap}, [\,])\ \mathsf{Cp}(\text{FMap}, \alpha)))\ .
\end{aligned}$$

We abbreviate the right hand side as $\Lambda(\mathsf{Cp}(\text{FMap}, \alpha) :: * \to *).\ t'$. We now define the **newtype** component as follows:

$$\textbf{newtype}\ \mathsf{Cp}(\text{FMap}, [\,]) = \Lambda(\mathsf{Cp}(\text{FMap}, \alpha) :: * \to *).\ \mathit{FMapList}\ t'\ .$$

This whole construct is now translated as a user-defined **newtype** statement would be. Therefore, $\mathsf{Cp}(\text{FMap}, [\,])$ is added to the kind environment, and an embedding-projection pair and structural representation for the type is generated. The resulting environment entries are returned.

We do not show the rule for **type** specialization requests as it is quite similar. The only difference is that a type synonym is generated for the component, of the form

$$\textbf{type}\ \mathsf{Cp}(T, T') = \{\Lambda b_i :: \kappa'_i.\}^{i \in 1..n}\ t'\ ,$$

which is then translated recursively.

16.7.7 Translation of declarations

We have sketched in Section 16.3 that it is also necessary to adapt the generation of components for generic functions slightly. Not only embedding-projection pairs for the component type have to be defined for the *bimap* wrapper, but also embedding-projection pairs for applications of type-indexed datatypes, at least if the components are not type synonyms, but defined via **newtype** or **data**.

$$\mathsf{cmbase}_{K;\Gamma}\bigl(x\ \langle\{\beta_i\}_{,}^{i\in 1..r}\mid\{\beta'_j\}_{,}^{j\in 1..s}\rangle\rightsquigarrow\{\gamma_i\sim T_i\}_{,}^{i\in 1..r'}\mid\{\gamma'_j\}_{,}^{j\in 1..s'}\bigr)\equiv A$$

$$\frac{\begin{array}{c}
\Theta\equiv\{\beta_i\}_{,}^{i\in 1..r}\mid\{\beta'_j\}_{,}^{j\in 1..s}\\
\mathsf{mbase}_{K;\Gamma}\bigl(x\ \langle\Theta\rangle\bigr)\equiv A\\
[\![A::\rho]\!]^{\mathrm{gftx}}_{K;\varepsilon;\bar{\Sigma};\Psi}\equiv A'\\
\rho\equiv\bigl(\{\{T_{i,k}\ \langle\beta_i\rangle::\kappa_{i,k}\}_{,}^{k\in 1..r_i}\}_{,}^{i\in 1..r}\\
\qquad\{\{,T'_{j,k}\ \langle\beta'_j\rangle::\kappa'_{j,k}\}^{k\in 1..s_j}\}^{j\in 1..s}\bigr)\Rightarrow *\\
\{\{\gamma_{i,k}\ \text{fresh}\}^{k\in 1..r_i}\}^{i\in 1..r}\qquad\{\{\gamma'_{j,k}\ \text{fresh}\}^{k\in 1..s_j}\}^{j\in 1..s}\\
A'\equiv\{\{\mathbf{Let\ type}\ T_{i,k}\ \langle\beta_i\rangle\ \{a_h\}^{h\in 1..\mathsf{arity}(\kappa_{i,k})}=\gamma_{i,k}\ \mathbf{In}\}^{k\in 1..r_i}\}^{i\in 1..r}\\
\qquad\{\{\mathbf{Let\ type}\ T'_{j,k}\ \langle\beta'_j\rangle\ \{a_h\}^{h\in 1..\mathsf{arity}(\kappa'_{j,k})}=\gamma'_{j,k}\ \mathbf{In}\}^{k\in 1..s_j}\}^{j\in 1..s}\ A\\
\Theta'\equiv\{\gamma_{i,k}\sim T_{i,k}\}_{,}^{k\in 1..r_i})\mid\{\{\gamma'_{j,k}\}_{,}^{k\in 1..s_j}\}_{,}^{i\in 1..s}
\end{array}}{\mathsf{cmbase}_{K;\Gamma;\bar{\Sigma};\Psi}\bigl(x\ \langle\Theta\rangle\rightsquigarrow\{\Theta'\}_{,}^{i\in 1..r}\bigr)\equiv A'}\ (\text{cmbase})$$

Figure 16.16: Closed base type with respect to applications of type-indexed types

We have given an example for *lookup* $\langle[\alpha]\rangle$, which would need to be defined as

$$\begin{array}{l}
\mathit{lookup}\ \langle[\alpha]\rangle=\\
\quad\mathbf{let}\ \mathit{bimap}\ \langle\beta\rangle=\mathsf{ep}([\,])\\
\quad\qquad\mathit{bimap}\ \langle\gamma\rangle=\mathsf{ep}(\mathsf{Cp}(\mathrm{FMap},[\,]))\\
\quad\mathbf{in}\ \ \mathit{to}\ \mathit{bimap}\ \langle\mathbf{Let\ type}\ \mathrm{FMap}\ \langle\beta\rangle\ (a::*)=\gamma\ \mathbf{In}\ \mathsf{mbase}(\mathit{lookup}\ \langle\beta\rangle)\rangle\\
\qquad\qquad(\mathit{lookup}\ \langle\mathsf{Str}([\,])\ \alpha\rangle)\ \ .
\end{array}$$

In a first step, we discuss the judgment of the form

$$\mathsf{cmbase}_{K;\Gamma}\bigl(x\ \langle\{\beta_i\}_{,}^{i\in 1..r}\mid\{\beta'_j\}_{,}^{j\in 1..s}\rangle\rightsquigarrow\{\gamma_i\sim T_i\}_{,}^{i\in 1..r'}\mid\{\gamma'_j\}_{,}^{j\in 1..s'}\bigr)\equiv A$$

which can be used to produce a closed base type application, by surrounding an application of mbase with local redefinitions on the type level, thereby filling in fresh dependency variables for applications of type-indexed datatypes.

The cmbase function thus takes

$$\mathsf{mbase}(\mathit{lookup}\ \langle\beta\rangle)$$

to

$$\mathbf{Let\ type}\ \mathrm{FMap}\ \langle\beta\rangle\ (a::*)=\gamma\ \mathbf{In}\ \mathsf{mbase}(\mathit{lookup}\ \langle\beta\rangle)$$

in the situation of the example.

At the same time, cmbase returns information about the **Let** constructs that have been added. We separate the dependency variables γ_i that are bound to type-level dependencies for the β_i, and the γ'_j that correspond to the β'_j. For the γ_i, we also return the name of the corresponding type-indexed type. In the example, we have $r \equiv 1$ and $s \equiv 0$. Hence, the function returns

$$\gamma \sim \text{FMap} \mid \varepsilon \ .$$

Even though syntactically not a proper type argument tuple, we abbreviate

$$\{\gamma_i \sim T_i\}_{,}^{i \in 1..r'} \mid \{\gamma'_j\}_{,}^{j \in 1..s'}$$

using the symbol Θ in the rules to come.

The rule (cmbase) for the cmbase judgment is shown in Figure 16.16: we use A as an abbreviation for the mbase call. The kind of A is ρ, and because A can contain applications of type-indexed datatypes, ρ can have dependency constraints for the dependency variables β_i and β'_j that occur in the mbase call. We make all these dependencies on the β_i and the β'_j explicit in ρ. The $T_{i,k}$ are all type-indexed datatypes that occur in dependencies with one of the β_i, and the $T'_{j,k}$ are all type-indexed datatypes that occur in dependencies with one of the β'_j.

We now introduce fresh dependency variables for each of the dependencies, and associate $\gamma_{i,k}$ with $T_{i,k}$ and $\gamma'_{j,k}$ with $T'_{j,k}$. The dependency variables are then used as right hand sides for type-level local redefinitions.

In the example, we get

$$A' \equiv \textbf{Let type } \text{FMap} \ \langle\beta\rangle \ (a_1 :: *) = \gamma_{1,1} \ \textbf{In} \ A \ .$$

In the local redefinitions, we have to introduce type variables – such as a_1 in the example, if the kind of the dependency is not $*$. For example, the base kind of FMap is of kind $* \to *$, therefore the dependency of A on FMap $\langle\beta\rangle$ is of kind $* \to *$ as well, thus we need to introduce one local type variable.

In general, the number of type variables introduced is $\mathsf{arity}(\kappa_{i,k})$ and $\mathsf{arity}(\kappa'_{j,k})$, where $\mathsf{arity}(\kappa)$ is defined as

$$\begin{aligned} &\mathsf{arity}(*) &&\equiv 0 \\ &\mathsf{arity}(\kappa_1 \to \kappa_2) &&\equiv 1 + \mathsf{arity}(\kappa_2) \ . \end{aligned}$$

Together with the locally redefined type argument A', we return the newly introduced dependency variables $\gamma_{i,k}$ (together with the name of the corresponding type indexed type) and $\gamma'_{j,k}$. In our example, that is just $\gamma_{1,1} \sim$ FMap.

Now, we can proceed to the generation of additional components for a generic function declaration, as described by rule (d/tr-tidgf) in Figure 16.17. This rule

$$[\![d_{\text{FCR+tif+par}} \rightsquigarrow \Gamma_2; \Sigma_2]\!]^{\text{gftx}}_{K;\Gamma_1;\Delta;\Sigma_1;\Psi} \equiv \{d_{\text{FCR}\ i}\}^{i \in 1..n}_{;}$$

$$\frac{\begin{array}{c} P_0 \equiv T\ \{\alpha_k\}^{k \in 1..\ell} \\ \mathsf{Str}(T) \equiv \{\Lambda a_k :: \kappa_k.\}^{k \in 1..\ell}\ t \in \Psi \\ K; \Gamma' \vdash^{tpsig} x\ \langle \pi \rangle :: \sigma \\ \pi \equiv (\{b_i :: *\}^{i \in 1..r} \mid \{b'_j :: \kappa'_j\}^{j \in 1..s}) \\ \{\beta_i \text{ fresh}\}^{i \in 1..r} \qquad \{\beta'_j \text{ fresh}\}^{j \in 1..s} \\ \mathsf{deptt}_{K;\Gamma}(x\ \langle \pi \rangle, x) \equiv \pi \\ K' \equiv K\ \{, \beta_i :: *\}^{i \in 1..r}\ \{, \beta'_j :: \kappa'_j\}^{j \in 1..s} \\ \Theta \equiv \{\beta_i\}^{i \in 1..r}_{,} \mid \{\beta'_j\}^{j \in 1..s}_{,} \qquad \Theta' \equiv \{\gamma_i \sim T_i\}^{i \in 1..r'}_{,} \mid \{\gamma'_j\}^{j \in 1..s'}_{,} \\ e_0 \equiv \textbf{let}\ \{bimap\ \langle \beta_i \rangle = \mathsf{ep}(T)\}^{i \in 1..r}_{;} \\ \{bimap\ \langle \beta'_j \rangle = EP\ id\ id\}^{j \in 1..s}_{;} \\ \{bimap\ \langle \gamma_i \rangle = \mathsf{ep}(\mathsf{Cp}(T_i, T))\}^{i \in 1..r'}_{;} \\ \{bimap\ \langle \gamma'_j \rangle = EP\ id\ id\}^{j \in 1..s'}_{;} \\ \textbf{in}\ \ to\ bimap\ \langle \mathsf{cmbase}_{K';\Gamma}(x\ \langle \Theta \rangle \rightsquigarrow \Theta') \rangle \\ (x\ \langle t\{[\alpha_k\ /\ a_k]\}^{k \in 1..\ell} \rangle) \\ d' \equiv x\ \langle a \rangle = \textbf{typecase}\ a\ \textbf{of}\ P_0 \to e_0\ \{; P_i \to e_i\}^{i \in 1..n} \\ [\![d' \rightsquigarrow \Gamma'; \Sigma']\!]^{\text{gftx}}_{K;\Gamma;\Delta;\Sigma;\Psi} \equiv \{d_{\text{FCR}\ i}\}^{i \in 1..n}_{;} \end{array}}{\begin{array}{c} [\![x\ \langle a \rangle = \textbf{typecase}\ a\ \textbf{of}\ \{P_i \to e_i\}^{i \in 1..n}_{;} \rightsquigarrow \Gamma'; \Sigma']\!]^{\text{gftx}}_{K;\Gamma;\Delta;\Sigma;\Psi} \\ \equiv \{d_{\text{FCR}\ i}\}^{i \in 1..n}_{;} \end{array}} \quad \text{(d/tr-tidgf)}$$

Figure 16.17: Translation of FCRT+gftx declarations to FCRT, replaces Figure 11.2

is a replacement for rule (d/tr-gf), defined in Figure 11.2 on page 183, and it is very similar. Nevertheless it is probably the most complicated rule in this thesis, because this is the place where type-indexed functions and type-indexed datatypes interact. The difference with (d/tr-gf) is that we refer to cmbase instead of mbase to generate the wrapper, and that we use the information about the introduced dependency variables for a local redefinition on the value level to plug in the embedding-projection pair for the appropriate components of the type-indexed datatypes.

In our running example, the call *lookup* $\langle [\alpha] \rangle$, we get

$$\begin{aligned}
e_0 \equiv \mathbf{let}\ & \mathit{bimap}\ \langle \beta_1 \rangle = \mathsf{ep}([\,]) \\
& \mathit{bimap}\ \langle \gamma_1 \rangle = \mathsf{ep}(\mathsf{Cp}(\mathrm{FMap}, [\,])) \\
\mathbf{in}\ \ & \mathit{to\ bimap}\ \langle \mathbf{Let\ type}\ \mathrm{FMap}\ \langle \beta_1 \rangle\ (a :: *) = \gamma_1\ \mathbf{In}\ \mathsf{mbase}(\mathit{lookup}\ \langle \beta_1 \rangle) \rangle \\
& \qquad (\mathit{lookup}\ \langle \mathsf{Str}([\,])\ \alpha_1 \rangle)\ ,
\end{aligned}$$

as promised. This expression will be added in an arm of the form

$$[\alpha_1] \rightarrow e_0$$

to the **typecase** construct that defines the *lookup* function.

Apart from the just discussed new rule (d/tr-tidgf), there is one more thing we have to verify while translating type-indexed functions: if a type-indexed function refers to a type-indexed datatype, all the types that are in the signature of the type-indexed datatype must also be in the signature of the type-indexed function. The reason is that we cannot generically generate a component for a function if the corresponding component of the type-indexed datatype is non-generic. This check is best added to the rule (d/tr-tif) which translates the final type-indexed function, including all generically generated components.

At this point, we conclude our account of the translation of type-indexed datatypes. We have omitted some less interesting parts, such as the translation of the environments or how to adapt the checking of the entire program.

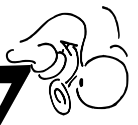
16.8 Translation by specialization and type-indexed datatypes

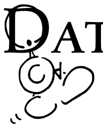
Translation by specialization in the presence of type-indexed datatypes requires a stronger interaction between different phases of the translation. The reason is that applications of type-indexed datatypes can occur nested in type arguments. As a consequence, the generation of structural representation types and embedding-projection pairs cannot be done once and for all in the beginning; the components of type-indexed datatypes are datatypes themselves, and their structure must be made available.

We therefore need a staged approach: a specialization request for a type argument involving other calls to type-indexed datatypes can only be processed after the components for those calls have been generated, and the structure of the associated datatypes is available.

Furthermore, specialization of type-indexed function calls should only take place after all components of type-indexed datatype have been generated. This allows us to use ordinary generic functions on type-indexed datatypes, too, so that we can for example compare two finite maps for equality using the generic application *equal* $\langle$FMap $\langle$Tree Int$\rangle$ String$\rangle$, or generate a generic string representation of a zipper using *show* $\langle$Loc $\langle$TreeF$\rangle\rangle$.

17 Alternative Views on Datatypes





In Chapter 10, we have shown that we can view a datatype as an isomorphic representation type, thereby replacing complex concepts such as *n*-ary choice and constructors with multiple arguments by simpler ones, and by using a limited set of abstract datatypes to embody these simple concepts. In particular, we have made use of three datatypes, Unit, Prod, and Sum. But was this choice a good one? Well, it certainly is not the only choice possible. In fact, we have already discussed in the context of the generic *bimap* function that it can be useful to treat the universal quantifier over a type variable of kind $*$, written Forall_*, as a type constructor $(* \to *) \to *$ (cf. Section 11.3). We have also noted that the treatment of identity types (cf. Section 11.4) might warrant the introduction of Id as a special type, or even a marker Type for the boundary of a datatype. Furthermore, we have mentioned that some datatypes can be represented as fixpoints in Section 8.4 and 12.3. Some functions, including generic functions, can be written more easily using such a representation of a datatype, because it allows explicit access to the points of recursion.

All the described changes are modifications or variations of the *standard view* that we have described in Chapter 10. These modifications require to adapt the translation of datatypes into generic representation types. In this chapter, we will discuss several possibilities for alternative views and the resulting consequences for the implementation.

We will start, in Section 17.1, with an extension of the structural representation that allows access to information about names of datatypes, constructors, and possibly record field labels. Using this extension, it is possible to write functions such as *read* and *show* generically. Afterwards, in Section 17.2, we investigate the implementation of a fixpoint view and sketch how it can be implemented. This view can be used to write a generic function *children* that collects the recursive children of a datatype. Sections 17.3, 17.4, and 17.5 list three more possible views: a balanced encoding of sums and products, a list-like encoding of sums and products, and a view which allows *constructor cases*, specific cases of generic functions for special behaviour on a single constructor. Finally, in Section 17.6, we point out that it would be nice if one could define one's own views as a user of the language.

17.1 Constructors and labels

Parsing and pretty-printing are common examples of generic functions: showing values on screen, storing them on disk in both human readable or compressed format, exchanging data between different programs – all these problems have one thing in common: a universal representation has to be found for all datatypes and we need functions to convert between values of the datatypes and the universal representation.

We have already discussed simple encoding and decoding functions, in Section 7.4. With *encode* a value of any type is stored in a list of bits, and, given the type of the expected value, *decodes* (or *decode* from Section 12.1), recovers the value from such a list.

All encodings that we can define so far are either not generic or cannot make use of the name information that is contained in a datatype, simply because the structural representation is not equipped to represent such names. For example, the datatypes

```
data Bool   = False     | True
data Bit    = 0         | 1
data Motion = Clockwise | Counterclockwise
```

all three have the same structural representation, namely Sum Unit Unit. Consequently, the encoding of [*False*, *True*, *True*] is the same as the encoding of [0, 1, 1]. While this might be desirable in some situations, for the purpose of conversion between isomorphic datatypes (Atanassow and Jeuring 2004), it is unwanted in general. Haskell provides a *show* function that generically computes a human-readable string representation of almost any Haskell value, and a counterpart *read* that can be used to recover a typed value from a string. The string representation contains the constructor names from the datatypes, and thereby ensures that the string representation of one value cannot be read back as a value of another type.

17.1.1 Extending structural representations

To be able to do the same with a Generic Haskell function, we must extend the structural representation of datatypes and give the programmer some means to access information about constructors and datatypes, such as their names.

As a first step, we introduce a new datatype along the lines of Unit, Sum, Prod, and Zero. The datatype Con, defined as

$$\mathbf{data}\ \mathrm{Con}\ (a :: *) = \mathit{Con}\ a\ ,$$

serves as a simple marker for the position of a constructor. We insert such tags in the structural representation between the sum and the product structure. For instance, the representation of any of the three types Bool, Bit, and Motion above becomes

$$\mathrm{Sum}\ (\mathrm{Con}\ \mathrm{Unit})\ (\mathrm{Con}\ \mathrm{Unit})\ ,$$

and the representation of lists becomes

$$\Lambda a :: *.\ \mathrm{Sum}\ (\mathrm{Con}\ \mathrm{Unit})\ (\mathrm{Con}\ (\mathrm{Prod}\ a\ [a]))\ .$$

While the representation types are generated, the compiler tags each occurrence of Con with an abstract value describing the original constructor, called the **constructor descriptor**. In the case of lists, the constructors are [ ] and (:), so the compiler remembers

$$\Lambda a :: *.\ \mathrm{Sum}\ (\mathrm{Con}_{[\,]}\ \mathrm{Unit})\ (\mathrm{Con}_{(:)}\ (\mathrm{Prod}\ a\ [a]))\ .$$

The tags are omitted in the actual translation, and ignored in the embedding-projection pair as well. The only place where they play a role is if a generic function has a specific arm for the type Con. The type pattern associated with the Con type takes a special form, not Con α, but

$$\mathrm{Con}\ x\ \alpha\ ,$$

the *x* being a value-level variable that is bound to the constructor descriptor. This value is of the abstract type ConDescr. This datatype supports several methods to obtain information about the constructor, among them

```
conName :: ConDescr → String
conType  :: ConDescr → String ,
```

to return the name of the constructor and the name of the datatype that the constructor belongs to.

17.1.2 Showing values

As a concrete example of the use of a constructor descriptor in a generic function, let us look at the function *showP*, a slightly generalized variant of Haskell's *show* that takes an additional argument of type String → String. This parameter is used internally to place parentheses around a fragment of the result when needed.

```
showP ⟨a :: ∗⟩ :: (showP ⟨a⟩) ⇒ (String → String) → a → String
showP ⟨Unit⟩        p Unit      = ""
showP ⟨Sum α β⟩  p (Inl x)    = showP ⟨α⟩ p x
showP ⟨Sum α β⟩  p (Inr x)    = showP ⟨β⟩ p x
showP ⟨Prod α β⟩ p (x₁ × x₂) = showP ⟨α⟩ p x₁ ++ " " ++ showP ⟨β⟩ p x₂
showP ⟨Con c α⟩   p (Con x)   = let parens x = "(" ++ x ++ ")"
                                      body     = showP ⟨α⟩ parens x
                                  in  if null body
                                      then conName c
                                      else  p (conName c ++ " " ++ body)
showP ⟨[α]⟩         p xs        = let body = ( concat ⟨[ ]⟩
                                                 · intersperse ", "
                                                 · map ⟨[ ]⟩ (showP ⟨α⟩ id))
                                                   xs
                                  in  "[" ++ body ++ "]"
```

The type Unit represents a constructor with no fields. In such a situation, the constructor name alone is the representation, and it will be generated from the Con case, so we do not need to produce any output here. We just descend through the sum structure; again, no output is produced here because the constructor names are produced from the Con case. A product concatenates fields of a single constructor; we therefore show both components, and separate them from each other by whitespace.

Most of the work is done in the arm for Con. We show the body of the constructor, using parentheses where necessary. The body is empty if and only if there are no fields for this constructor. In this case, we only return the name of the constructor. Here we make use of the function *conName* on the constructor descriptor *c* to obtain that name. Otherwise, we connect the constructor name and the output of the body with a space, and surround the result with parentheses if requested by parameter *p*.

The last case is for lists and implements the Haskell list syntax, with brackets and commas.

In addition to the cases above, we need cases for abstract primitive types such as Char, Int, or Float that implement the operation in some primitive way.

The function *show* is defined in terms of *showP* via generic abstraction, instantiating the first parameter to the identity function, because outer parentheses are usually not required.

$$\begin{array}{l} \mathit{show}\ \langle a :: *\rangle\ :: (\mathit{showP}\ \langle a\rangle) \Rightarrow a \to \text{String} \\ \mathit{show}\ \langle \alpha :: *\rangle = \mathit{showP}\ \langle \alpha\rangle\ \mathit{id} \end{array}$$

The definition of a generic *read* function that parses the generic string representation of a value is also possible using the Con case, and only slightly more involved because we have to consider partial consumption of the input string and possible failure.

17.1.3 Generic cardinality

There are other generic functions that can benefit from constructor information being available in the structural representation of datatypes. For example, we can define a variant of the cardinality function *card* that terminates even for infinite types. We define a new datatype

$$\textbf{data}\ \text{Cardinality} = \mathit{Fin}\ \text{Int} \mid \mathit{Inf}$$

and expect *card* to return a value of that type, thus *Inf* in the case that it is called on a datatype with infinitely many elements.

The trick is that we can break infinite recursion by maintaining a list of datatypes we have visited:

$$\mathit{card}'\ \langle a :: *\rangle :: (\mathit{card}'\ \langle a\rangle) \Rightarrow [\text{String}] \to \text{Cardinality}\ \ .$$

The argument list is supposed to contain all the type names that we already have visited on our path down. The definition of the function is as follows:

$$\begin{array}{ll} \mathit{card}'\ \langle \text{Int}\rangle & \mathit{visited} = \mathit{Inf} \\ \mathit{card}'\ \langle \text{Float}\rangle & \mathit{visited} = \mathit{Inf} \end{array}$$

```
card′ ⟨Char⟩       visited = fromEnum (maxBound :: Char)
                              − fromEnum (minBound :: Char) + 1
card′ ⟨Zero⟩       visited = Fin 0
card′ ⟨Unit⟩       visited = Fin 1
card′ ⟨Sum α β⟩    visited = card′ ⟨α⟩ visited ⊕ card′ ⟨β⟩ visited
card′ ⟨Prod α β⟩   visited = card′ ⟨α⟩ visited ⊗ card′ ⟨β⟩ visited
card′ ⟨α → β⟩      visited = card′ ⟨β⟩ visited ⊚ card′ ⟨α⟩ visited
card′ ⟨Con c α⟩    visited
    | conType c ∈ visited = Inf
    | otherwise           = card′ ⟨α⟩ (conType c : visited) .
```

We consider the base types Int and Float as infinite (although both are not really), and define the cardinality of Char using the Bounded class in Haskell. The Zero type contains no elements, the Unit type contains one element. For sums, products, and functions, we make use of the operations ⊕, ⊗, and ⊚, which are versions of addition, multiplication, and exponentiation, that work on the Cardinality datatype. Their definitions are given in Figure 17.1. If a constructor is encountered, we check if the associated datatype is already known to us. If that is the case, we can return *Inf* for this part of the structure. If not, we descend, but add the new datatype to the list. Note that this approach considers all recursive types as infinite, which is not correct. For example, the type

```
data Rec = Rec Rec
```

which has no values not relying on lazy evaluation, is wrongly identified as an infinite type.

We can define a wrapper around *card′*, using generic abstraction, that initializes the list of known datatypes to the empty list:

```
card ⟨a :: ∗⟩  :: (card′ ⟨a⟩) ⇒ Cardinality
card ⟨α :: ∗⟩ = card′ ⟨α⟩ [] .
```

Note that *card* is an example of a generic function where the type argument does not occur at all in the type (except for the type argument itself). Neither does it take any arguments, and is more a generic *value* than a generic function.

17.1.4 An additional view?

We have seen now how to write generic functions using constructor information provided that we have the Con marker. We have discussed the adapted translation to structural representations, and also the adapted specialization code that passes the constructor descriptor whenever a Con component is called. But how

$$
\begin{array}{llll}
(\oplus),(\otimes),(\circledwedge) & & & :: \text{Cardinality} \to \text{Cardinality} \to \text{Cardinality} \\
\mathit{Inf} & \oplus\, x & & = \mathit{Inf} \\
x & \oplus\, \mathit{Inf} & & = \mathit{Inf} \\
\mathit{Fin}\ x_1 & \oplus\, \mathit{Fin}\ x_2 & & = \mathit{Fin}\ (x_1 + x_2) \\
\mathit{Inf} & \otimes\, \mathit{Fin}\ 0 & & = \mathit{Fin}\ 0 \\
\mathit{Inf} & \otimes\, x & & = \mathit{Inf} \\
\mathit{Fin}\ 0 & \otimes\, \mathit{Inf} & & = \mathit{Fin}\ 0 \\
x & \otimes\, \mathit{Inf} & & = \mathit{Inf} \\
\mathit{Fin}\ x_1 & \otimes\, \mathit{Fin}\ x_2 & & = \mathit{Fin}\ (x_1 {*} x_2) \\
\mathit{Inf} & \circledwedge\, \mathit{Fin}\ 0 & & = \mathit{Fin}\ 1 \\
\mathit{Inf} & \circledwedge\, x & & = \mathit{Inf} \\
\mathit{Fin}\ 0 & \circledwedge\, \mathit{Inf} & & = \mathit{Fin}\ 0 \\
x & \circledwedge\, \mathit{Inf} & & = \mathit{Inf} \\
\mathit{Fin}\ x_1 & \circledwedge\, \mathit{Fin}\ x_2 & & = \mathit{Fin}\ \left(x_1^{x_2}\right)
\end{array}
$$

Figure 17.1: Operations on the Cardinality type

do these additions integrate with the generic functions that do not care about constructors?

Fortunately, the integration is very simple. With a minor addition, we can transparently make use of constructor information: if the function does not need it, an explicit arm can be omitted, i.e., the function can be written exactly as before. If the function does need it, it is available.

To achieve this, we always produce the enriched structural representation, containing the Con markers. But other than Unit, Sum, Prod, and Zero, we do not treat Con itself as an abstract type, but rather *define* a structural representation for it, namely

$$\Lambda a :: *.\ a\ \ .$$

In other words, $\mathsf{Str}(\text{Con}\ t) \equiv t$ for any t. Now we can proceed with specialization as we are used to.

17.1.5 Record labels

Haskell facilitates the definition of records by providing a special syntax for datatypes, where field names are associated with the fields of constructors. These field names can also be made accessible for generic functions such as *show*, in an analogous way to the treatment of constructor information.

We define a new datatype Lab as

$$\textbf{data}\ \text{Lab}\ (a :: *) = \mathit{Lab}\ a$$

and use Lab to mark the position of a field label. Field label markers are thus inserted in the product structure of a type. Internally, they are again tagged with an abstract description of the label, in this case of type LabDescr. This **label descriptor** is then used in components for the type Lab.

17.2 Fixpoints of regular functors

The idea of a fixpoint view on datatypes is that regular datatypes are automatically represented as fixpoints of their pattern functors. Regular datatypes are the class of datatypes that generic (or polytypic) functions work on in PolyP (Jansson and Jeuring 1997). A datatype of the form

$$\textbf{data}\ T \quad = \{\Lambda a_i :: \kappa_i.\}^{i \in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}$$

is called **regular** if all recursive references on the right hand side – i.e., in the $t_{j,k}$ – have the form $T\ \{a_i\}^{i \in 1..n}$. Then, we can define the **pattern functor** of T to be

$$\begin{aligned}\textbf{data}\ \mathsf{PF}(T) = {} & \{\Lambda a_i :: \kappa_i.\}^{i \in 1..\ell}\ \Lambda c :: *. \\ & \{\mathsf{PF}(C_j)\ \{t_{j,k}[c\ /\ T\ \{a_i\}^{i \in 1..\ell}]\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}\ ,\end{aligned}$$

where c is a fresh type variable. The type $T\ \{a_i\}^{i \in 1..\ell}$ is isomorphic to the type $\text{Fix}\ (\mathsf{PF}(T)\ \{a_i\}^{i \in 1..\ell})$, assuming the usual definition of Fix as

$$\textbf{data}\ \text{Fix}\ (f :: * \to *) = \mathit{In}\ (f\ (\text{Fix}\, f))\ .$$

For example, for the datatype of lists $[\,]$, we get

$$\begin{aligned}\textbf{data}\ \mathsf{PF}([\,])\ (a :: *)\ (c :: *) = {} & \mathsf{PF}([\,]) \\ \mid {} & \mathsf{PF}((:))\ a\ c\ ,\end{aligned}$$

and the isomorphism between $[a]$ and $\text{Fix}\ (\mathsf{PF}([a]))$ is witnessed by

$$\begin{aligned}&\mathsf{epf}([\,]) :: \forall a :: *.\ \text{EP}\ [a]\ (\text{Fix}\ (\mathsf{PF}([a])))\\ &\mathsf{epf}([\,]) = \textbf{let}\ \begin{aligned}[t] &\mathit{from}\ [\,] && = \mathit{In}\ (\mathsf{PF}([\,]))\\ &\mathit{from}\ (x : xs) && = \mathit{In}\ (\mathsf{PF}((:))\ x\ (\mathit{from}\ xs))\\ &\mathit{to}\quad (\mathit{In}\ (\mathsf{PF}([\,]))) && = [\,]\\ &\mathit{to}\quad (\mathit{In}\ (\mathsf{PF}((:))\ x\ xs)) && = x : \mathit{to}\ xs\end{aligned}\\ &\qquad\qquad\ \textbf{in}\ \ \mathit{EP}\ \mathit{from}\ \mathit{to}\ \ .\end{aligned}$$

Note that while the pattern functor $\mathsf{PF}([\,])$ is not recursive, the conversion functions *from* and *to* that make up the embedding-projection pair are.

For datatypes of higher-rank kinds, we need access to the *bimap* function to define embedding-projection pairs between a datatype and its fixpoint representation.

The pattern functor is itself translated into a structural representation, this time using the standard sum of products view and an associated embedding-projection pair.

$$
\begin{array}{l}
\mathsf{Str}(\mathsf{PF}([\,])) \equiv \text{Sum Unit (Prod } a\ c)\\
\mathsf{ep}(\mathsf{PF}([\,]))\ ::\ \forall a :: *.\ \text{EP}\ \mathsf{PF}([\,])\ \mathsf{Str}(\mathsf{PF}([\,]))\\
\mathsf{ep}(\mathsf{PF}([\,])) = \textbf{let}\ \begin{array}[t]{lll} \mathit{from} & (\mathsf{PF}([\,])) & = \mathit{Inl}\ \text{Unit}\\ \mathit{from} & (\mathsf{PF}((:))\ x\ y) & = \mathit{Inr}\ x\ y\\ \mathit{to} & (\mathit{Inl}\ \text{Unit}) & = \mathsf{PF}([\,])\\ \mathit{to} & (\mathit{Inr}\ x\ y) & = \mathsf{PF}((:))\ x\ y \end{array}\\
\qquad\qquad\qquad\ \ \textbf{in}\ \ \mathit{EP}\ \mathit{from}\ \mathit{to}\ \ .
\end{array}
$$

Using the fixpoint view, one can for example write the generic function *children*:

$$
\begin{array}{l}
\mathit{children}\ \langle a :: *\rangle\ \ ::\ (\mathit{collect}\ \langle a \mid a\rangle) \Rightarrow a \to [a]\\
\mathit{children}\ \langle \text{Fix}\ \gamma\rangle = \textbf{let}\ \mathit{collect}\ \langle\alpha\rangle\ x = [x]\\
\qquad\qquad\qquad\qquad\ \textbf{in}\ \ \mathit{collect}\ \langle\gamma\ \alpha\rangle\ \ .
\end{array}
$$

This function depends on *collect* and uses it to gather all direct recursive occurrences of the datatype in a list. Using this definition, the call

$$\mathit{children}\ \langle[\text{Int}]\rangle\ [3,4,5]$$

evaluates to $[[4,5]]$, because non-empty lists have exactly one child, their tail. The call

$$\mathit{children}\ \langle\text{Tree Int}\rangle\ \big(\mathit{Node}\ (\mathit{Node}\ \mathit{Leaf}\ 2\ \mathit{Leaf})\ 3\ (\mathit{Node}\ \mathit{Leaf}\ 5\ \mathit{Leaf})\big)$$

returns $[(\mathit{Node}\ \mathit{Leaf}\ 2\ \mathit{Leaf}), (\mathit{Node}\ \mathit{Leaf}\ 5\ \mathit{Leaf})]$.

Of course, we can define *children* without a fixpoint view, but then we cannot directly apply it to datatypes such as [Int] or Tree Int. It is then the programmer's responsibility to explicitly transform the data structures into a fixpoint representation, prior to calling *children*.

By defining a type-indexed datatype to hold algebras belonging to a datatype, we can also define a more convenient variant of *cata* (cf. Section 12.3):

$$
\begin{array}{l}
\text{AlgebraOf}\ \langle a :: *\rangle\qquad\quad\ ::\ (\text{ID}\ \langle a\rangle) \Rightarrow *\\
\textbf{type}\ \text{AlgebraOf}\ \langle\text{Fix}\ \gamma\rangle\ r = \text{ID}\ \langle\gamma\rangle\ r \to r
\end{array}
$$

$$
\begin{array}{l}
\mathit{cata}\ \langle a :: *\rangle \qquad :: (\mathit{map}\ \langle a,a\rangle) \Rightarrow \forall (r :: *).\ \mathrm{AlgebraOf}\ \langle a\rangle\ r \to a \to r \\
\mathit{cata}\ \langle \mathrm{Fix}\ \gamma\rangle\ \mathit{alg} = \mathit{alg} \cdot \mathit{map}\ \langle\gamma\rangle\ (\mathit{cata}\ \langle \mathrm{Fix}\ \gamma\rangle\ \mathit{alg}) \cdot \mathit{out}\ \ .
\end{array}
$$

The definition of *cata* is as before, but the first argument to the cata function is now of the type-indexed type AlgebraOf, which is also defined using the fixpoint view, and maps a datatype to a function from its pattern functor to the result type.

We can define a generic abstraction on *cata* specifically for functors of kind $* \to *$:

$$
\begin{array}{ll}
\mathit{cataf}\ \langle c :: * \to *\rangle & :: (\mathit{map}\ \langle c,c\rangle) \Rightarrow \forall (a :: *)\ (r :: *). \\
 & \qquad\qquad\qquad \mathrm{AlgebraOf}\ \langle c\rangle\ a\ r \to c\ a \to r \\
\mathit{cataf}\ \langle \gamma :: * \to *\rangle & = \textbf{let}\ \mathit{map}\ \langle\alpha\rangle = \mathit{id}\ \textbf{in}\ \mathit{cata}\ \langle \gamma\ \alpha\rangle\ \ .
\end{array}
$$

In the type signature, we have used “short notation” for local redefinition on the type level. Without short notation, we would have to write

$$
\begin{array}{ll}
\mathit{cataf}\ \langle c :: * \to *\rangle & :: (\mathit{map}\ \langle c,c\rangle) \Rightarrow \forall (a :: *)\ (r :: *). \\
 & \qquad\qquad\qquad \textbf{Let}\ \mathrm{ID}\ \langle\alpha\rangle = a \\
 & \qquad\qquad\qquad \textbf{In}\ \ \mathrm{AlgebraOf}\ \langle c\ \alpha\rangle\ r \to c\ a \to r\ \ .
\end{array}
$$

We can further facilitate the use of *cata* by defining an operator that lets us construct algebras easily:

$$
\begin{array}{l}
(\oplus) \qquad :: \forall (a :: *)\ (b :: *)\ (r :: *) \to (a \to r) \to (b \to r) \to (\mathrm{Sum}\ a\ b \to r) \\
(\oplus)\ f_1\ f_2 = \lambda x \to \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\qquad \mathit{Inl}\ x_1 \to f_1\ x_1 \\
\qquad\qquad\qquad \mathit{Inr}\ x_2 \to f_2\ x_2\ \ .
\end{array}
$$

We assume that $\oplus$ associates to the right. Using both $\oplus$ and *cataf*, one can now apply catamorphisms with almost the same convenience as by using Haskell’s *foldr* function. The expression

$$
\mathit{cataf}\ \langle[\,]\rangle\ (\mathit{const}\ 0 \oplus \lambda(x \times y) \to x + y)\ [1\,..\,100]
$$

evaluates to 5050, and

$$
\begin{array}{l}
\mathit{cataf}\ \langle \mathrm{Tree}\rangle\ \big(\mathit{const}\ [\,] \oplus \lambda(x \times (y \times z)) \to y : x +\!\!+\, z\big) \\
\qquad\qquad\quad \big(\mathit{Node}\ (\mathit{Node}\ \mathit{Leaf}\ 1\ \mathit{Leaf})\ 2\ (\mathit{Node}\ \mathit{Leaf}\ 3\ \mathit{Leaf})\big)
\end{array}
$$

results in $[2,1,3]$.

17.3 Balanced encoding

The standard view on datatypes uses a binary representation for sums and products. The choice between multiple constructors is expressed as a nested, right-deep, binary sum. Similarly, multiple fields are translated to a nested, right-associative, binary product. These encodings are somewhat arbitrary: we could have used a left-deep nesting instead, or a balanced encoding.

The direction datatype

data Direction = *North* | *East* | *South* | *West*

that is represented as

Sum Unit (Sum Unit (Sum Unit Unit))

using the standard view, would be translated to

Sum (Sum Unit Unit) (Sum Unit Unit)

in a balanced encoding.

The choice of encoding can affect both efficiency and behaviour of generic functions. The generic equality function *equal*, if applied to datatypes with many constructors, is less efficient using the standard view than it would be with the balanced encoding. If both values belong to the rightmost constructor, then the whole sum structure must be traversed in the standard view, whereas we have a logarithmic complexity for the balanced encoding.

This difference is even more visible in the *encode* function, where it leads to different results. We need a maximum of three bits to encode a value of type Direction in the standard view, but two bits are always sufficient using the balanced view.

17.4 List-like sums and products

Yet another possibility to represent sums and products is to use a right-biased nesting, but to always use the neutral elements Zero and Unit to mark the end of a sum or product, respectively. The neutral elements play a similar role as the "nil" constructor [ ] for lists. The encoding of the Direction type would become

Sum Unit (Sum Unit (Sum Unit (Sum Unit Zero))) ,

and the type of rose trees, defined as

$$\textbf{data}\ \mathrm{Rose}\ (a :: *) = \mathit{Fork}\ a\ [\mathrm{Rose}\ a]\ ,$$

would be embedded as

$$\mathrm{Sum}\ \big(\mathrm{Prod}\ a\ (\mathrm{Prod}\ [\mathrm{Rose}\ a]\ \mathrm{Unit})\big)\ \mathrm{Zero}\ \ .$$

This encoding has the advantage of being more uniform than the standard view: there are always as many occurrences of Sum as there are constructors, and as many occurrences of Prod per constructor as there are fields. The left argument of a Sum always represents a single constructor, and the left argument of a Prod always represents a single field.

If a generic function should operate in a certain uniform way for all fields of a constructor, then that function might be easier to write using the list-like view.

17.5 Constructor cases

The "Generic Haskell, specifically" paper (Clarke and Löh 2003) describes constructor cases, which are a way to view constructors themselves as datatypes and use them in type patterns during the definition of type-indexed functions. For example, the function *freecollect*, defined on page 218 in Section 14.1, could be written in the following way using constructor cases:

```
freecollect ⟨a :: ∗⟩ :: (freecollect ⟨a⟩) ⇒ a → [Var]
freecollect extends termcollect
freecollect ⟨Lam⟩ (Lam (v,t) e)    = filter (≠ v) (freecollect ⟨Expr⟩ e)
freecollect ⟨Let⟩  (Let  (v,t) e e′) = freecollect ⟨Expr⟩ e
                                      ++ filter (≠ v) (freecollect ⟨Expr⟩ e′)  .
```

We define special behaviour for the *Lam* and *Let* constructors of the Expr datatype simply by writing additional arms. The constructors which are not mentioned still enjoy generic behaviour. The original definition defines an arm for the Expr datatype itself, using a **case** construct to detect the interesting constructors:

```
freecollect ⟨Expr⟩ e =
  case e of
    Lam (v,t) e′     → filter (≠ v) (freecollect ⟨Expr⟩ e′)
    Let  (v,t) e′ e″ → freecollect ⟨Expr⟩ e′
                          ++ filter (≠ v) (freecollect ⟨Expr⟩ e″)
    _                → termcollect ⟨Expr⟩ e  .
```

Using constructor cases has a couple of subtle advantages over the approach using the **case** construct. The first is that we do not need to specify the fallback explicitly. The line *termcollect* ⟨Expr⟩ *e* is easy to write in this particular case. It is, however, more difficult to do so if we are not extending an already existing function using a default case, or if the type pattern of the case we define involves dependency variables, and the dependencies of the fallback functions need to be redirected, such as described in the conversion function for default cases (cf. Section 14.4).

The other advantage requires to take a look of how constructor cases are implemented. In the presence of constructor cases, datatypes are encoded in two levels. The datatype for expressions, which is defined as

```
data Expr = Var  Var
          | App  Expr Expr
          | Lam  (Var, Type) Expr
          | Let  (Var, Type) Expr Expr ,
```

is expressed as a sum of its constructors:

```
Str(Expr) ≡ Sum (Con Con(Var))
                (Sum (Con Con(App))
                     (Sum (Con Con(Lam))
                          (Con Con(Let)))) ,
```

where Con is an internal function taking a constructor name to a type name. The constructor types are type synonyms pointing to the datatype they belong to:

```
type Con(Var)  = Expr
type Con(App)  = Expr
type Con(Lam)  = Expr
type Con(Let)  = Expr .
```

However, these type synonyms have non-standard structural representations, in that they are translated into only the product structure of the fields of the constructor they represent:

```
Str(Var)  ≡ Var
Str(App)  ≡ Prod Expr Expr
Str(Lam)  ≡ Prod (Var, Type) Expr
Str(Let)  ≡ Prod (Var, Type) (Prod Expr Expr) .
```

Using this encoding, the standard specialization procedure from Chapter 11 can be used to translate constructor cases. As can be seen from the example, the type synonyms representing the constructor types are positioned between sum and product structure, and beneath the Con markers from Section 17.1. This means that if a constructor case is used, the functionality for that the type-indexed function implements for the sum structure or the Con case is still implemented for the associated datatype. In the definition of *freecollect* without constructor cases, where the Expr datatype is explicitly specified, the functionality of *freecollect* for Sum and Con is no longer executed for values of type Expr that belong to the *Lam* or *Let* constructors. For *freecollect*, this is no real disadvantage, because no Con definition exists, and the Sum case only propagates the collected variables.

17.6 A language for views

The examples discussed in this chapter have demonstrated that some generic functions are easier to define or more efficient using a non-standard view on datatypes, and some functions cannot even be defined in the standard view from Chapter 10.

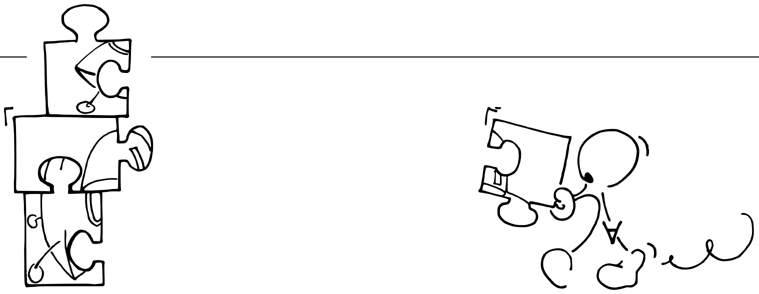
In some situations, such as in the case of constructor information (cf. Section 17.1), it is possible to extend the standard view in such a way that functions which do not require the extra information continue to work without modification. Nevertheless, additional constructors appear in the encoding and thus in the translation, which makes the generated code less efficient. Other views, such as the balanced encoding or the list-like view, are not extensions, but proper variations of the encoding.

In general, a **view** defines a way to translate a datatype into a type expression, the *structural representation*, that is (at least nearly) isomorphic to the original datatype. An embedding-projection pair is required to witness the conversion. Generic functions that make use of that particular view can then be defined by providing cases for the types that occur in the structural representations generated by the view.

Generic Haskell views are related to Wadler's views (1987), proposed as an extension for the Haskell language. In Wadler's proposal, views provide different ways of viewing one datatype, by providing different sets of data constructors by which a value of the datatype may be constructed or deconstructed. In this chapter, we show how to view datatypes in various ways, by providing different sets of type constructors by which structural representations of datatypes may be constructed or deconstructed.

Because there are many views (our list of examples certainly is not exhaustive), and because it is unlikely that there is one single best view, it seems desirable to equip Generic Haskell with a language to define views. The user would then need to specify for each generic function which view it should use. How exactly this might be achieved is the subject of ongoing research. It is difficult to guarantee that a user-defined view is correct. If the representations that are produced are not truly isomorphic, run-time errors might be the result. It is also unclear how different views interact: can functions that are defined using different views all seamlessly depend on each other?

18 Modules



Real programs rarely consist of just one huge blob of code. Haskell offers the possibility to group functions into modules. Modules can be compiled separately, and code from one module can be reused in several others.

All the concepts in Generic Haskell were demonstrated in the limited setting of single-module programs. All of them can easily be transferred to a language with multiple modules. However, if we allow multiple modules, this opens up a few new questions and problems: how can type-indexed functions and type-indexed types be exported from one and imported into another module, and, probably more important, how does separate compilation work in the presence of type-indexed entities and genericity?

In this chapter, we will try to answer these questions. To this end, we add modules both to the language FCRT, resulting in FCRTM, and to FCRT+gftx, yielding FCRTM+gftx, and discuss how to translate the latter into the former. We will not go into the details of the semantics of FCRTM or the module system itself, but concentrate on the additional overhead resulting from type-indexed functions and datatypes.

The new languages are introduced in Section 18.1. The translation of the language FCRTM+gftx to FCRTM is the topic of Section 18.2. In the end, we discuss two

Programs			
P	$::= \{B\}_{;}^{i \in 1..n}$	$(n \in 1\,..)$	
			sequence of modules
Modules			
B	$::= H$ **where** $\{D_j;\}^{j \in 1..n}$ **main** $= e$		
			type declarations plus main expression
Module headers			
H	$::=$ **module** M $(\{X_i\}_{,}^{i \in 1..m})$ $(\{I_j\}_{;}^{j \in 1..n})$		
			header with export list
Import declarations			
I	$::=$ **import** M		import module by name
Export list entries			
X	$::= x$		function name
	$\mid$ **abstract** T		abstract type
	$\mid$ T		concrete type

Figure 18.1: Language with modules FCRTM, extends Figures 3.1, 3.12, and 15.1

problems that become more pronounced in the presence of modules: whether type-indexed functions should be open or closed, in Section 18.4, and why explicit specialization requests for type-indexed datatypes are necessary, in Section 18.3.

18.1 A language with modules

Figure 18.1 shows the syntax extensions necessary to cover programs consisting of multiple modules, defining language FCRTM as an extension of the functional core language with recursive let and all type declaration constructs. Compared to Haskell, there are some modifications – mainly simplifications – because we want to concentrate on the issues related to type-indexed entities.

A program consists of a non-empty sequence of modules now. The last module is considered the main module. The structure of modules is similar to the former syntax of programs. There is now a module header, but the body of a module still consists of declarations that are followed by a main expression. The main expression of the main module is to be evaluated; all other main expressions are type checked, but discarded.

A module header declares a name for a module, and governs which names are exported and imported. We allow to hide functions and types by not including

Export list entries			
X	$::=$	$\ldots$	everything from Figure 18.1
	$\mid$	**abstract** $T\ \langle R\rangle$	abstract type-indexed type component
	$\mid$	$T\ \langle R\rangle$	concrete type-indexed type component

Figure 18.2: Language with modules and type-indexed entities FCRTM+gftx, extends Figure 18.1

them in the export list. A module can only be included as a whole, and since we are not interested in namespace management here, we do not consider qualified imports or the renaming of imported modules.

Consequently, an import declaration simply lists a number of modules. The export list controls the entities that are visible in other modules that import the current module: functions and values can be exported by adding them to the list; for datatypes, there is a choice: a datatype can be exported as an abstract type (i.e., without knowledge of constructors and structure) or as a concrete type (including constructors and its structure). It is illegal to export a type synonym as abstract type.

While all datatypes that are declared in a module can also be exported, functions can only be exported if they are defined at the **toplevel** of the module. The set of toplevel functions is the set of functions defined in the outermost **let** construct of the main expression. We therefore require the main expression to be a **let** construct, but since the declaration part may be empty, this is not a restriction.

We do not go into the details of the semantics. The idea is that all exported entities of a module M are visible in all following modules that import module M. For simplicity, we assume that an imported function can only be typed if all necessary datatypes are also in scope, at least as abstract types. This is different from Haskell, where datatypes can be internally available to help type checking imported functions, yet not be in scope, thus being invisible to the user.

To type a module, we must transfer knowledge from the imported modules: the kind environment contains information about all imported datatypes and type synonyms, both abstract and concrete; the type environment contains knowledge about all imported functions, and the constructors of imported concrete datatypes. Furthermore, we need the definitions of imported type synonyms in the type synonym environment. Of course, we must also produce this information for the exported entities while processing a module.

Let us now look at the additional modifications needed to define language FCRTM+gftx, having modules as well as type-indexed functions and datatypes, all at the same time. Syntactically, we require almost no changes. The only difference is in the export lists: function names can now also be names of toplevel generic functions, and we allow to export a type-indexed datatype concrete or abstract (we discuss in the next section what that means precisely). In addition, we need the possibility to export specialized components of type-indexed datatypes – such as generated by explicit specialization requests – both abstract and concrete.

18.2 Translation of modules containing type-indexed entities

In this section, we describe how to translate an FCRTM+gftx module, i.e., a module possibly containing type-indexed functions and type-indexed datatypes, into an FCRTM module.

This will inevitably also say some things about the module semantics for normal functions and datatypes, as they are part of the extended language as well.

As far as module syntax is concerned, the export list must be simplified. Occurrences of type-indexed functions, type-indexed datatypes, and type-indexed datatype components have to be replaced. For this purpose, we introduce a judgment of the form

$$[\![X_{\text{FCRTM+gftx}} \rightsquigarrow \mathrm{K}_2; \Gamma_2; \Sigma_2; \bar{\Sigma}_2; \Psi_2; \mathrm{E}_2]\!]^{\text{mod}}_{\mathrm{K}_1;\Gamma_1;\Sigma_1;\bar{\Sigma}_1;\Psi_1;\mathrm{E}_1} \equiv \{X_{\text{FCRTM}\,i}\}_{,}^{i\in 1..n} \quad .$$

Under all input environments, we process a single export list entry X_1. This results in contributions to the environments that will be exported from the current module, and a number of export list entries in the target program.

The translation rules are shown in Figures 18.3 and 18.4. The rule (x-fun) is for ordinary functions. A function x mentioned in the export list must be a toplevel function of the current module or imported from somewhere else. This is checked by testing that x has an entry in the global type environment Γ that is provided as input. This test also verifies that x is not type-indexed. The resulting program has the same entry x in its export list. Furthermore, we return a type environment Γ' containing an entry for x, because this is the information we export to other modules.

For a type-indexed function x, the rule (x-tif) checks whether an appropriate type signature $x\ \langle\vartheta\rangle :: \sigma$ is contained in the input type environment Γ. We extract the signature of x from the input signature environment Σ, using the signature function explained on page 222 in Section 14.4. Both the type signature and the signature of x are then exported to other modules through the output environments Γ' and Σ'. The translated module will not contain the type-indexed function anymore, though. Instead, it contains the components that have been generated for the function. Therefore, the export list of the target module contains entries of the form $\mathsf{cp}(x, T_i)$ for all types T_i that constitute the signature of x.

$$\llbracket X_{\text{FCRTM+gftx}} \rightsquigarrow \mathrm{K}_2; \Gamma_2; \Sigma_2; \bar{\Sigma}_2; \Psi_2; \mathrm{E}_2 \rrbracket^{\text{mod}}_{\mathrm{K}_1;\Gamma_1;\Sigma_1;\bar{\Sigma}_1;\Psi_1;\mathrm{E}_1} \equiv \{X_{\text{FCRTM}\,i}\}_{,}^{i \in 1..n}$$

$$\frac{x :: t \in \Gamma \qquad \Gamma' \equiv x :: t}{\llbracket x \rightsquigarrow \varepsilon; \Gamma'; \varepsilon; \varepsilon; \varepsilon; \varepsilon \rrbracket^{\text{mod}}_{\Gamma} \equiv x} \quad \text{(x-fun)}$$

$$\frac{\begin{array}{c} x\ \langle \vartheta \rangle :: \sigma \in \Gamma \qquad \Gamma' \equiv x\ \langle \vartheta \rangle :: \sigma \\ \mathsf{signature}_{\Sigma}(x) \equiv \{T_i\}_{,}^{i \in 1..n} \qquad \Sigma' \equiv \{x\ \langle T_i \rangle\}_{,}^{i \in 1..n} \end{array}}{\llbracket x \rightsquigarrow \varepsilon; \Gamma'; \Sigma'; \varepsilon; \varepsilon; \varepsilon \rrbracket^{\text{mod}}_{\Gamma;\Sigma} \equiv \{\mathsf{cp}(x, T_i)\}_{,}^{i \in 1..n}} \quad \text{(x-tif)}$$

$$\frac{T :: \kappa \in \mathrm{K} \qquad \mathrm{K}' \equiv T :: \kappa}{\llbracket \textbf{abstract}\ T \rightsquigarrow \mathrm{K}'; \varepsilon; \varepsilon; \varepsilon; \varepsilon; \varepsilon \rrbracket^{\text{mod}}_{\mathrm{K}} \equiv \textbf{abstract}\ T} \quad \text{(x-atp)}$$

$$\frac{\begin{array}{c} T :: \kappa \in \mathrm{K} \qquad \mathrm{K}' \equiv T :: \kappa \\ \Gamma' \equiv \mathsf{constructors}_{\Gamma}(T) \\ \mathsf{Str}(T) \equiv \{\Lambda a_i :: \kappa_i .\}^{i \in 1..n}\ t \in \Psi \qquad \Psi' \equiv (\mathsf{Str}(T) \equiv \{\Lambda a_i :: \kappa_i .\}^{i \in 1..n}\ t) \\ \mathsf{ep}(T) = e \in \mathrm{E} \qquad \mathrm{E}' \equiv \mathsf{ep}(T) = e \end{array}}{\llbracket T \rightsquigarrow \mathrm{K}'; \Gamma'; \varepsilon; \varepsilon; \Psi'; \mathrm{E}' \rrbracket^{\text{mod}}_{\mathrm{K};\Gamma;\Psi;\mathrm{E}} \equiv T, \mathsf{ep}(T)} \quad \text{(x-tp)}$$

$$\frac{\begin{array}{c} T\ \langle a \rangle :: \bar{\sigma} \in \mathrm{K} \qquad \mathsf{Signature}_{\bar{\Sigma}}(T) \equiv \{T_i\}_{,}^{i \in 1..n} \\ \left\{\llbracket T\ \langle T_i \rangle \rightsquigarrow \mathrm{K}'_i; \Gamma'_i; \varepsilon; \bar{\Sigma}'_i; \Psi'_i; \mathrm{E}'_i \rrbracket^{\text{mod}} \equiv \{X_{i,j}\}_{,}^{j \in 1..m_i}\right\}^{i \in 1..n} \\ \mathrm{K}' \equiv T\ \langle \vartheta \rangle :: \bar{\sigma}\ \{, \mathrm{K}'_i\}^{i \in 1..n} \qquad \Gamma' \equiv \{\Gamma'_i\}_{,}^{i \in 1..n} \qquad \bar{\Sigma}' \equiv \{\bar{\Sigma}'_i\}_{,}^{i \in 1..n} \\ \Psi' \equiv \{\Psi'_i\}_{,}^{i \in 1..n} \qquad \mathrm{E}' \equiv \{\mathrm{E}'_i\}_{,}^{i \in 1..n} \end{array}}{\llbracket T \rightsquigarrow \mathrm{K}'; \Gamma'; \varepsilon; \bar{\Sigma}'; \Psi'; \mathrm{E}' \rrbracket^{\text{mod}}_{\mathrm{K};\bar{\Sigma}} \equiv \left\{\{X_{i,j}\}_{,}^{j \in 1..m_i}\right\}_{,}^{i \in 1..n}} \quad \text{(x-tid)}$$

Figure 18.3: Translation of FCRTM+gftx export list entries to FCRTM, continued from Figure 18.3

$$\frac{T\ \langle \mathsf{Join}(R)\rangle \in \bar{\Sigma} \qquad \bar{\Sigma}' \equiv T\ \langle \mathsf{Join}(R)\rangle \qquad [\![\mathbf{abstract}\ \mathsf{Cp}(T,\mathsf{Join}(R)) \rightsquigarrow \mathrm{K}';\varepsilon;\varepsilon;\varepsilon;\varepsilon;\varepsilon]\!]^{\mathrm{mod}} \equiv \{X_i\}_{,}^{i\in 1..n}}{[\![T\ \langle R\rangle \rightsquigarrow \mathrm{K}';\varepsilon;\varepsilon;\bar{\Sigma}';\varepsilon;\varepsilon]\!]^{\mathrm{mod}}_{\bar{\Sigma}} \equiv \{X_i\}_{,}^{i\in 1..n}} \quad \text{(x-atc)}$$

$$\frac{T\ \langle \mathsf{Join}(R)\rangle \in \bar{\Sigma} \qquad \bar{\Sigma}' \equiv T\ \langle \mathsf{Join}(R)\rangle \qquad [\![\mathsf{Cp}(T,\mathsf{Join}(R)) \rightsquigarrow \mathrm{K}';\Gamma';\varepsilon;\varepsilon;\Psi';\mathrm{E}']\!]^{\mathrm{mod}} \equiv \{X_i\}_{,}^{i\in 1..n}}{[\![T\ \langle R\rangle \rightsquigarrow \mathrm{K}';\Gamma';\varepsilon;\bar{\Sigma}';\Psi';\mathrm{E}']\!]^{\mathrm{mod}}_{\bar{\Sigma}} \equiv \{X_i\}_{,}^{i\in 1..n}} \quad \text{(x-tc)}$$

Figure 18.4: Translation of FCRTM+gftx export list entries to FCRTM, continued from Figure 18.3

The next rule, (x-atp), handles abstract types. If a type T is to be exported abstract, it has to be in scope, i.e., there must be an entry in K. The form of this entry implies that T is not type-indexed. The export list entry then appears in the same form in the resulting module, and the kind information of the type (but not its constructors, nor structural representation, nor embedding-projection pair) is exported.

In contrast, if a type T is exported as a concrete type, then rule (x-tp) determines the constructors of type T from the type environment Γ, using an internal operation of the form

$$\mathsf{constructors}_{\Gamma}(T) \equiv \Gamma' \ .$$

This operation traverses Γ for all constructors of which the result type matches T, and returns the filtered environment. Furthermore, the structural representation is extracted from the type synonym environment Ψ, and the embedding-projection pair from E. The kind information of T, the types of the constructors, the structural representation, and the embedding-projection pair are all exported, hence available for other modules. The target module does also contain the type T, therefore the target export list exports T as well. Furthermore, the translated module contains the embedding-projection pair as a toplevel declaration, and $\mathsf{ep}(T)$ is exported with its type.

For a type-indexed type T, the rule (x-tid) checks if a kind signature is present in K. For type-indexed types, we can define a function Signature in analogy to signature for type-indexed functions: the statement

$$\mathsf{Signature}_{\bar{\Sigma}}(T) \equiv \{T_i\}_{,}^{i\in 1..n}$$

expresses that the T_i are the named types for which the user has explicitly defined the type-indexed type T. This function requires that we mark each entry

$$[\![B_{\text{FCRTM+gftx}} :: t \rightsquigarrow \mathrm{K}_2;\Gamma_2;\Sigma_2;\Psi_2;\mathrm{E}_2;\{M_k\}_{,}^{k\in 1..\ell}]\!]^{\text{mod}}_{\mathrm{K}_1;\Gamma_1;\Sigma_1;\Psi_1;\mathrm{E}_1} \equiv B_{\text{FCRTM}}$$

$$\frac{\begin{array}{c} B \equiv H\ \textbf{where}\ \{D_i\}_{;}^{i\in 1..n}\ \textbf{main} = e \\ \{[\![D_i \rightsquigarrow \mathrm{K}_i;\Gamma_i;\bar{\Sigma}_i;\Psi_i;\mathrm{E}_i]\!]^{\text{gftx}}_{\mathrm{K};\bar{\Sigma};\Psi} \equiv \{D_{i,j}\}_{;}^{j\in 1..\ell_i}\}^{i\in 1..n} \\ \mathrm{K} \equiv \mathrm{K}_0\ \{,\mathrm{K}_i\}^{i\in 1..n} \qquad \Gamma \equiv \Gamma_0\ \{,\Gamma_i\}^{i\in 1..n} \\ \bar{\Sigma} \equiv \bar{\Sigma}_0\ \{,\bar{\Sigma}_i\}^{i\in 1..n} \qquad \Psi \equiv \Psi_0\ \{,\Psi_i\}^{i\in 1..n} \qquad \mathrm{E} \equiv \{\mathrm{E}_i\}_{,}^{i\in 1..n} \\ [\![e :: t \rightsquigarrow \Gamma_{\text{toplevel}}]\!]^{\text{mod}}_{\mathrm{K};\Gamma;\Sigma;\bar{\Sigma};\Psi} \equiv \textbf{let}\ \{d'_i\}_{;}^{i\in 1..m'}\ \textbf{in}\ e' \\ [\![H \rightsquigarrow \mathrm{K}';\Gamma';\Sigma';\Psi';\mathrm{E}';\{M_k\}_{,}^{k\in 1..\ell}]\!]^{\text{mod}}_{\mathrm{K};\Gamma,\Gamma_{\text{toplevel}};\Sigma;\Psi;\mathrm{E}_0,\mathrm{E}} \equiv H' \\ B' \equiv H'\ \textbf{where}\ \{\{D_{i,j}\}_{;}^{j\in 1..\ell_i}\}_{;}^{i\in 1..n}\ \textbf{main} = \textbf{let}\ \mathrm{E}\ \{;d'_i\}^{i\in 1..m'}\ \textbf{in}\ e' \end{array}}{[\![B :: t \rightsquigarrow \mathrm{K}';\Gamma';\Sigma';\Psi';\mathrm{E}';\{M_k\}_{,}^{k\in 1..\ell}]\!]^{\text{mod}}_{\mathrm{K}_0;\Gamma_0;\Sigma_0;\Psi_0;\mathrm{E}_0} \equiv B'} \quad \text{(p-module)}$$

Figure 18.5: Translation of FCRTM+gftx modules to FCRTM

in $\bar{\Sigma}$ whether it stems from an explicit definition or from a generically generated component (cf. Section 14.4). Each of the components is then translated on its own. The results from those translations, together with the kind signature for the whole datatype, are then exported.

A request for a type component is handled by rules (x-atc) and (x-tc), depending on whether it is for an abstract component or not. In both cases, we verify that the component exists by checking the type-level signature environment $\bar{\Sigma}$. We also export this signature entry to other modules. Furthermore, we translate an export list entry for the name of the associated type component and return the results.

In addition to the translation of the export list, we have to create a translation rule to handle complete modules. This rule is based on (p-gprog) on page 181.

The judgment for modules is of the form

$$[\![B_{\text{FCRTM+gftx}} :: t \rightsquigarrow \mathrm{K}_2;\Gamma_2;\Sigma_2;\Psi_2;\mathrm{E}_2;\{M_k\}_{,}^{k\in 1..\ell}]\!]^{\text{mod}}_{\mathrm{K}_1;\Gamma_1;\Sigma_1;\Psi_1;\mathrm{E}_1} \equiv B_{\text{FCRTM}} \ .$$

It takes a module and environments as input. We assume that there is a rule for programs consisting of several modules that analyzes the import lists of the modules and provides the imported entities in their respective environments. Thus, we assume that all input environments contain precisely those entities from other modules that are visible in $B_{\text{FCRTM+gftx}}$, plus possible built-in entities. All entities that are exported from the module are returned, together with the module names that constitute the import list. Furthermore, a type t is assigned to the main expression of the module.

The rule (p-module) is shown in Figure 18.5. The module B is of the form

$$H\ \mathbf{where}\ \{D_i\}_{;}^{i\in 1..n}\ \mathbf{main} = e\ .$$

There is the module header H, containing import and export information. Then there are type declarations D_i, and the main expression e.

Each of the type declarations D_i is checked and translated. For this, we can reuse the rules from Chapter 16 on type-indexed datatypes. Each declaration can produce potentially multiple FCRTM declarations. In addition, several entries for various environments can be produced. All type declarations may be mutually recursive, therefore we pass to the type declarations environments K, $\bar{\Sigma}$, and Ψ that contain not only the imported entities, but also the locally defined types.

The main expression is translated using an adapted judgment of the form

$$[\![e :: t \rightsquigarrow \Gamma_{\text{toplevel}}]\!]^{\text{mod}}_{K;\Gamma;\Sigma;\bar{\Sigma};\Psi} \equiv e'\ .$$

We will make a brief excursion now to explain how the main expression is translated (which is not shown in a figure), and return to the discussion of rule (p-module) thereafter. There a two differences between the judgment of the main expression here, and the previous judgments for expressions. First, we force the source expression e into the form of a **let** statement, to reveal the toplevel declarations of the expression. Hence, we can assume

$$e \equiv \{d_i\}_{;}^{i\in 1..m}\ .$$

The judgment returns the type environment Γ_{toplevel}, containing the types of the toplevel declarations d_i. Second, before the actual translation of the expression e, it will be extended by additional declarations:

$$\mathbf{let}\ \{d_{\text{gfcomps}\,i}\}_{;}^{i\in 1..m'}\ \{;d_i\}^{i\in 1..m}\ \mathbf{in}\ e\ .$$

For each imported type-indexed function x, we add one declaration. Each of these declarations is an empty **typecase** construct for the imported type-indexed function x:

$$x\ \langle a\rangle = \mathbf{typecase}\ a\ \mathbf{of}\ \varepsilon\ ,$$

with the exception that the statement does not shadow the existing definition, but extends it. Using this mechanism, we can make use of rule (d/tr-tidgf) (see page 269, Section 16.7.7) to locally (i.e., per module) generate additional generic components for the type-indexed functions.

For a type-indexed function in the context of modules, only the components that make up the signature of the function are exported. Generically generated components are always local to the current module. If the same component is needed in different modules, it is generated several times. This duplicates work, but spares us the burden of administrating the information which specializations are available already and which are not. A sophisticated compiler could optimize here. Note that the implementations of type-indexed functions need not be exported; the type information and the signature are sufficient for rule (d/tr-tidgf) to produce new components.

For type-indexed datatypes, we generate components generically only upon a specialization request. If these generated components are then exported, they can also be used in other modules. This is discussed in more detail in Section 18.3.

Back to (p-module): we thus translate the main expression e using the just discussed adapted translation rule, yielding the translated FCRTM expression

$$\mathbf{let}\ \{d'_i\}_{;}^{i\in 1..m'}\ \mathbf{in}\ e'$$

and the toplevel environment Γ_{toplevel}. We keep the toplevel of the translation separate, for reasons that will become obvious soon.

The module header H is translated using another judgment that we will not present here, because it is a trivial wrapper around the export list translation of Figures 18.3 and 18.4. The judgment is of the form

$$[\![H \rightsquigarrow \mathrm{K}_1; \Gamma_1; \Sigma_1; \Psi_1; \mathrm{E}_1; \{M_k\}_{,}^{k\in 1..\ell}]\!]^{\text{mod}}_{\mathrm{K}_2;\Gamma_2;\Sigma_2;\Psi_2;\mathrm{E}_2} \equiv H' \ .$$

Next to traversing the export list with the input environments and merging all the result environments, the judgment also returns the module names M_k from the import list.

The header translation is here called with environments containing all imported entities, plus all entries derived from the type declarations, plus all local toplevel declarations. The resulting environments are also the resulting environments for the module judgment.

It remains to construct the target module B' from the translated header H', the translated type declarations $D_{i,j}$. The local embedding-projection pairs, contained in E, are added to the toplevel declarations of the translated expressions. It is important that they are added to the toplevel, because rule (x-tp) on page 293 causes embedding-projection pairs to be exported if the associated type is exported as a concrete type, and only toplevel declarations are allowed for export.

18.3 Explicit specialization of type-indexed types is necessary

Type-indexed types have to be specialized explicitly by the user, by stating *specialization requests* in the program. This is different from type-indexed functions, where the components that are needed are inferred automatically from the calls that occur in the scope of a function.

One of the reasons for the explicit specialization of type-indexed types – the one mentioned in Chapter 16 – was that there is a choice to be made, namely whether a type should be specialized as a type synonym or as a new, distinct, datatype. In the presence of modules, there is one additional reason why it is very helpful to specialize type-indexed types explicitly.

As we have seen, a specialization of a type-indexed type to a type synonym is often not possible, because of the limitations that type synonyms have, mainly that they do not allow recursion. On the other hand, the specialization via **newtype** introduces a new datatype, one that is distinct from all other types that are in scope. As a consequence, generating the same component of a type-indexed type *twice* via a **newtype** statement also introduces two *different* types automatically.

$$\begin{tikzcd}
 & M_0 \quad \text{(defines } T_1\text{)} & \\
\text{(uses } T_1\ \langle T_2 \rangle\text{)} \quad M_1 \arrow[ur] & & M_2 \quad \text{(uses } T_1\ \langle T_2 \rangle\text{)} \arrow[ul] \\
 & M_3 \quad \text{(conflict!)} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 18.6: Conflicting components of type-indexed types

Now consider the following situation, shown in Figure 18.6. We have a module M_0, which defines a type-indexed type T_1. We want to apply type T_1 to another, recursive type T_2, in two other modules, M_1 and M_2. Therefore these two modules import T_1 from module M_0. Because T_2 is a recursive type, the component $\mathsf{Cp}(T_1, T_2)$ has to be a **newtype** component. This **newtype** definition cannot – in general – be placed in module M_0. The type T_2 might not even be known in that module. The component also cannot be contained in whatever module defines T_2, because there, the type-indexed type might not be known.

If we adopt the same technique that we use for type-indexed functions, we would thus generate the component twice, in the modules M_1 and M_2. But $T_1\ \langle T_2 \rangle$ in module M_1 is now a *different* type than $T_1\ \langle T_2 \rangle$ in module M_2! If we import both M_1 and M_2 into yet another module M_3, we will get a name clash – or, if we allow qualified imports, we will at least have to deal with two different types and have no (easy) way to convert from one to the other.

Explicit specializations provide a way out of this dilemma: the user decides where (and how often) the specialization $T_1\ \langle T_2 \rangle$ is performed and $\mathsf{Cp}(T_1, T_2)$ is

$$\begin{tikzcd}
 & & M_0 & (\text{defines } T_1) & \\
 & & M_{\text{spec}} \arrow[u] & (\text{explicitly specializes } T_1\ \langle T_2\rangle) & \\
(\text{uses } T_1\ \langle T_2\rangle) & M_1 \arrow[uur, dashed] \arrow[ur] & & M_2 \arrow[uul, dashed] \arrow[ul] & (\text{uses } T_1\ \langle T_2\rangle) \\
 & & M_3 \arrow[ul] \arrow[ur] & (\text{no conflict}) &
\end{tikzcd}$$

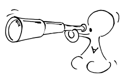
Figure 18.7: Explicit specialization in a separate module

generated. If the programmer instructs the compiler to perform the specialization $T_1\ \langle T_2\rangle$ in both M_1 and M_2, we will be in the same situation as before. But there is another option, depicted in Figure 18.7: one can define a *specialization module* M_{spec}, importing the type-indexed type T_1 from M_0 (and possibly the datatype T_2 from some other module), and containing at least the following code

```
module M_spec (T_1 ⟨T_2⟩) (import M_0; ...) where
newtype T_1 ⟨T_2⟩ as C_spec  .
```

This specialization module can be imported into both M_1 and M_2, and thus the two usage sites of $T_1\ \langle T_2\rangle$ can make use of the *same* component. In this situation, also further usage of the type in M_3 is unproblematic, because both imports will refer to the same type.

18.4 Open versus closed type-indexed functions



We have treated definitions of type-indexed functions (and of type-indexed datatypes) as *closed*. All arms of the **typecase** construct have to be given at the definition site of the function. It is not possible to add new cases at a later point.

This is not much of a problem if we consider programs consisting of only one module. If we discover that we need an additional case for an already existing type-indexed function, we can just go back to the definition and add it. However, in real programs consisting of many modules this may be very inconvenient.

A datatype for which we need a special case might not yet be defined in the module where the type-indexed function is defined, and we might not want the module containing the type-indexed function to depend on a lot of modules providing datatype definitions. Even worse, the type-indexed function might be part of a library, and if we change the library, we need access to the source, and the modification must be allowed by the license. Even then, we are in danger of losing the change again if the library is updated.

It would be much more convenient to be able to have an open type-indexed function, one that could be extended with new cases over different modules. This is a central feature of Haskell type classes: instances for classes can be defined in every module where the class is in scope.

Default cases (cf. Chapter 14) give us the possibility to extend type-indexed functions even though they remain closed. We can define a copy of the function, and extend the copy with new cases. Other functions that are defined in terms of the original function, however, retain a dependency on the original function, and have to be copied as well in order to work with the new definition.

Therefore, it might be desirable to allow open type-indexed functions that can be extended like type classes. Open type-indexed functions, do, however, also have a number of disadvantages. First, the order of cases in a **typecase** construct should not matter. In this thesis, we have only considered type patterns for which this property holds. This has been mainly for reasons of simplicity. It might be desirable to extend the pattern language to allow nested, and hence overlapping type patterns, too. For instance, Chakravarty and Keller (2003) describe a type-indexed datatype for arrays that performs a flattening optimization step if the type argument is of a specific, nested form.

For type classes, some Haskell implementations circumvent the problem by allowing *overlapping instances*. This does, however, lead to a rather complex and sometimes counterintuitive algorithm to find the correct instance in a given situation.

Another disadvantage of open type-indexed functions is that new cases can change the behaviour of already written code, but only to a certain extent. Assume we have a module M_1 which contains the following definitions:

$$\begin{aligned}
&\textbf{module}\ M_1\ (x, \mathit{test}_1, \mathit{test}_2)\ (\ldots)\ \textbf{where}\\
&x\ \langle a :: *\rangle && :: (x\ \langle a\rangle) \Rightarrow \mathrm{Int}\\
&x\ \langle \mathrm{Unit}\rangle && = 1\\
&x\ \langle \mathrm{Sum}\ \alpha\ \beta\rangle && = x\ \langle\alpha\rangle + x\ \langle\beta\rangle\\
&\mathit{test}_1\ \langle a :: *\rangle && :: (x\ \langle a\rangle) \Rightarrow \mathrm{Int}\\
&\mathit{test}_1\ \langle \alpha :: *\rangle && = 2 \cdot x\ \langle\alpha\rangle\\
&\mathit{test}_2 && :: \mathrm{Int}\\
&\mathit{test}_2 && = 2 \cdot x\ \langle \mathrm{Bool}\rangle\ .
\end{aligned}$$

The type-indexed function x can derive a generic component for type Bool, because cases for types Unit and Sum are defined. In the context of module M_1, the calls $test_1 \langle \text{Bool} \rangle$ and $test_2$ both evaluate to 4.

If we extend function x in another module M_2 as follows,

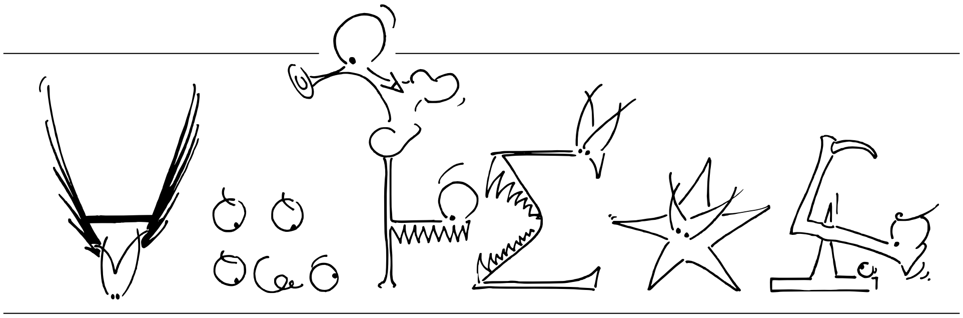
$$\begin{array}{l}\textbf{module}\ M_2\ (\ldots)\ (\textbf{import}\ M_1;\ldots)\ \textbf{where}\\ x\ \langle \text{Bool} \rangle \qquad = 0\ ,\end{array}$$

then, in the context of this module, call $test_1 \langle \text{Bool} \rangle$ evaluates to 0, whereas $test_2$ still evaluates to 4. The reason is, that $test_1$ still depends on x, and a component for x on type Bool, which is different for modules M_1 and M_2, is passed to the translation. For $test_2$, the component for x on type Bool is already determined and applied in module M_1, and cannot be changed later.

Other semantics of the above situation would be possible, but require deeper changes to the translation mechanism.

All in all, it might be a good compromise to allow both open and closed type-indexed functions in the language. By default, a type-indexed function would be closed. If it should be open, it must be marked using an **open** keyword. We might then impose certain restrictions on the functions (with respect to the type patterns, for example), but in turn allow the definition of new cases in other modules. Vytiniotis *et al.* (2004) also consider a calculus for generic programming in which both open and closed definitions are possible. The precise implications of such an extension for Generic Haskell remain future work.

Syntax overview



Complete syntax

Here, we list the complete syntax of the language, with all modifications that have been introduced. For each construct, we state where it has been defined.

Programs

$P \quad ::= \{D_i;\}^{i\in 1..n}\ \mathbf{main} = e$

[type declarations plus main expression, Figure 3.1]

$\quad\mid\ \ \{B\}_{;}^{i\in 1..n} \quad (n \in 1\,..)$

sequence of modules, Figure 18.1

Modules

$B \quad ::= H\ \mathbf{where}\ \{D_j;\}^{j\in 1..n}\ \mathbf{main} = e$

type declarations plus main expression, Figure 18.1

Module headers

$H \quad ::= \mathbf{module}\ M\ (\{X_i\}_{,}^{i\in 1..m})\ (\{I_j\}_{;}^{j\in 1..n})$

header with export list, Figure 18.1

Import declarations

$I \quad ::= \mathbf{import}\ M$ import module by name, Figure 18.1

Export list entries

X	$::= x$	function name, Figure 18.1
	$\mid$ $\mathbf{abstract}\ T$	abstract type, Figure 18.1
	$\mid$ T	concrete type, Figure 18.1
	$\mid$ $\mathbf{abstract}\ T\ \langle R\rangle$	abstract type-indexed type component, Figure 18.2
	$\mid$ $T\ \langle R\rangle$	concrete type-indexed type component, Figure 18.2

Type declarations

D	$::= \mathbf{data}\ T = \{\Lambda a_i :: \kappa_i.\}^{i\in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{\vert}^{j\in 1..m}$	datatype declaration, Figure 3.1
	$\mid$ $\mathbf{newtype}\ T = \{\Lambda a_i :: \kappa_i.\}^{i\in 1..\ell}\ C\ t$	newtype declaration, Figure 15.1
	$\mid$ $\mathbf{type}\ T = \{\Lambda a_i :: \kappa_i.\}^{i\in 1..\ell}\ t$	type synonym declaration, Figure 15.1
	$\mid$ $T\ \langle a\rangle = \mathbf{Typecase}\ a\ \mathbf{of}\ \{P_i \to t_i\}_{;}^{i\in 1..n}$	type-indexed datatype declaration, Figure 16.2
	$\mid$ $\mathbf{newtype}\ T\ \langle R\rangle\ \mathbf{as}\ C$	newtype specialization request, Figure 16.2
	$\mid$ $\mathbf{type}\ T\ \langle R\rangle$	type specialization request, Figure 16.2
	$\mid$ $T\ \langle \alpha :: \kappa\rangle = t$	type-level generic abstraction, Figure 16.2

Value declarations

d	$::= x = e$	function declaration, Figure 3.1
	$\mid$ $x\ \langle a\rangle = \mathbf{typecase}\ a\ \mathbf{of}\ \{P_i \to e_i\}_{;}^{i\in 1..n}$	type-indexed function declaration, Figure 4.1
	$\mid$ $x\ \langle \alpha\ \{(\gamma_i :: \kappa_i)\}^{i\in 1..n}\rangle = e$	local redefinition, Figure 8.3
	$\mid$ $x\ \langle \alpha :: \kappa\rangle = e$	generic abstraction, Figure 12.1
	$\mid$ $x'\ \mathbf{extends}\ x\ \mathbf{where}\ \{y_k\ \mathbf{as}\ y'_k\}_{;}^{k\in 1..\ell}$	default case, Figure 14.1

Kinds

κ	$::= *$	kind of manifest types, Figure 3.1
	$\mid$ $\kappa_1 \to \kappa_2$	functional kind, Figure 3.1

Kind signatures

$\bar{\sigma}$	$::= (\{T_k\}_{,}^{k\in 1..n}) \Rightarrow \kappa$	kind signature of type-indexed datatype, Figure 16.2

Qualified kinds

ρ	$::= (\bar{\Delta}) \Rightarrow \kappa$	qualified kind, Figure 16.2

Kind constraint sets

$\bar{\Delta}$	$::= \{\bar{Y}_i\}_{,}^{i\in 1..n}$	kind constraint set, Figure 16.2

Kind constraints

$\bar{Y} \quad ::= T\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: \rho_0$

kind dependency constraint, Figure 16.2

Types

$t, u ::= a, b, c, f, \ldots$	type variable, Figure 3.1
$\mid\ T$	named type, Figure 3.1
$\mid\ (t_1\ t_2)$	type application, Figure 3.1
$\mid\ \forall a :: \kappa . t$	universal quantification, Figure 3.1
$\mid\ T\ \langle A \rangle$	type-level generic application, Figure 16.2
$\mid\ \mathbf{Let}\ T\ \langle \alpha\ \{(\gamma_i :: \kappa_i)\}^{i \in 1..n} \rangle = t_1)\ \mathbf{In}\ t_2$	type-level local redefinition, Figure 16.2

Type patterns

$P \quad ::= T$	[named type pattern, Figure 4.1]
$\mid\ T\ \{\alpha_i\}^{i \in 1..n}$	parametrized named type pattern, Figure 6.1

Type arguments

$A \quad ::= T$	named type, Figure 4.1
$\mid\ \alpha, \beta, \gamma, \ldots$	dependency variable, Figure 6.1
$\mid\ (A_1\ A_2)$	type application, Figure 6.1
$\mid\ T\ \langle A \rangle$	type-level generic application, Figure 16.2
$\mid\ \mathbf{Let}\ T\ \langle \alpha\ \{(\gamma_i :: \kappa_i)\}^{i \in 1..n} \rangle = A_1)\ \mathbf{In}\ A_2$	type-level local redefinition, Figure 16.2

Qualified types

$q \quad ::= \{\forall a_i :: \kappa_i .\}^{i \in 1..n}\ (\Delta) \Rightarrow t$

qualified type, Figure 6.1

Constraint sets

$\Delta \quad ::= \{Y_i\}_{,}^{i \in 1..n}$ constraint set, Figure 6.1

Type constraints

$Y \quad ::= x\ \langle \alpha_0\ \{(\alpha_i :: \kappa_i)\}^{i \in 1..n} \rangle :: q$

dependency constraint, Figure 6.1

Type signatures

$\sigma \quad ::= (\{y_k\}_{,}^{k \in 1..n}) \Rightarrow t$

[type signature of type-indexed function, Figure 6.1]

$\mid\ (\{y_k\ \langle \vartheta_k \rangle\}_{,}^{k \in 1..n}) \Rightarrow t$

type signature of type-indexed function, Figure 9.8

Type tuple patterns

$\pi \quad ::= \{a_i :: *\}_{,}^{i \in 1..r} \mid \{b_j :: \kappa_j\}_{,}^{j \in 1..s}$

[type tuple pattern, Figure 9.4]

	$\mid\ \{a_i :: \kappa\}_{,}^{i \in 1..r} \mid \{b_j :: \kappa_j\}_{,}^{j \in 1..s}$	
		type tuple pattern, Figure 12.1

Type tuples

ϑ	$::= \{a_i\}_{,}^{i \in 1..r} \mid \{t_j\}_{,}^{j \in 1..s}$	
		type tuple, Figure 9.4

Type argument tuples

Θ	$::= \{A_i\}_{,}^{i \in 1..r} \mid \{t_j\}_{,}^{j \in 1..s}$	
		type argument tuple, Figure 9.4

Specialization request patterns

R	$::= T$	named type, Figure 16.2
	$\mid\ T\ \langle R \rangle$	type-indexed datatype component, Figure 16.2

Expressions

e	$::= x, y, z, \ldots$	variable, Figure 3.1
	$\mid\ C$	constructor, Figure 3.1
	$\mid\ (e_1\ e_2)$	application, Figure 3.1
	$\mid\ \lambda x \to e$	lambda abstraction, Figure 3.1
	$\mid\ \textbf{case}\ e_0\ \textbf{of}\ \{p_i \to e_i\}_{;}^{i \in 1..n}$	
		case, Figure 3.1
	$\mid\ \textbf{let}\ d\ \textbf{in}\ e$	[let, Figure 3.1]
	$\mid\ \textbf{fix}\ e$	[fixed point, Figure 3.1]
	$\mid\ \textbf{fail}$	run-time failure, Figure 3.8
	$\mid\ \textbf{letrec}\ \{d_i\}_{;}^{i \in 1..n}\ \textbf{in}\ e$	
		recursive let, Figure 3.12
	$\mid\ x\ \langle A \rangle$	generic application, Figure 4.1

Patterns

p	$::= C\ \{p_i\}^{i \in 1..n}$	constructor pattern, Figure 3.1
	$\mid\ x, y, z, \ldots$	variable pattern, Figure 3.1

Values

v	$::= C\ \{v_i\}^{i \in 1..n}$	constructor, Figure 3.8
	$\mid\ \lambda x \to e$	function, Figure 3.8
	$\mid\ \textbf{fail}$	run-time failure, Figure 3.8

Weak head-normal form

w	$::= C\ \{e_i\}^{i \in 1..n}$	constructor, Figure 3.8
	$\mid\ \lambda x \to e$	function, Figure 3.8
	$\mid\ \textbf{fail}$	run-time failure, Figure 3.8

All languages

Here, we provide a list of all languages that are used.

Languages without generic programming features (target languages)	
FC	functional core language
FCR	FC with recursive let
FCRT	FCR with **newtype** and **type**
FCRTM	FCRT with modules
Languages with generic programming features (source languages)	
FCR+tif	FCR with type-indexed functions
FCR+tif+par	FCR+tif with parametrized type patterns
FCR+tif+par+lr	FCR+tif+par with local redefinition
FCR+tif+mpar	FCR+tif+par with generalized type signatures
FCR+gf	FCR+tif+(m)par+lr with generic components
FCR+gf+gabs	FCR+gf with generic abstraction
FCR+gf+gabs+dc	FCR+gf+gabs with default cases
FCRT+gf+gabs+dc	like FCR+gf+gabs+dc, but based on FCRT
FCRT+gftx	FCRT+gf+gabs+dc with type-indexed datatypes
FCRTM+gftx	like FCRT+gftx, but based on FCRTM

Metavariables used

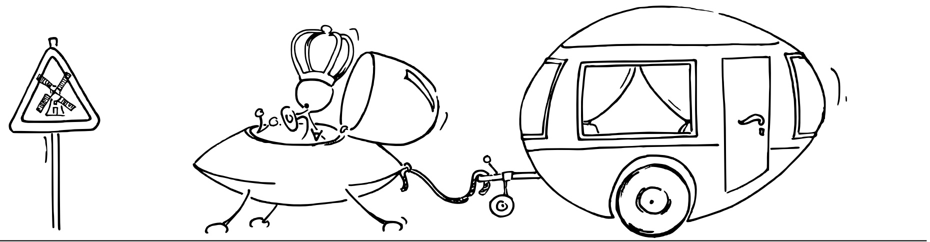
This is an exhaustive list of all metavariables that are used for entities in the core languages and their several extensions. For each metavariable, we list what it is used for and in which figure it is introduced.

a	type variable	Figure 3.1
b	type variable	Figure 3.1
c	type variable	Figure 3.1
d	value declaration	Figure 3.1
e	expression	Figure 3.1
f	type variable	Figure 3.1
i	natural number	
j	natural number	
k	natural number	
ℓ	natural number	
m	natural number	
n	natural number	
p	pattern	Figure 3.1

q	qualified type	Figure 6.1
r	natural number	
s	natural number	
t	type	Figure 3.1
u	type	Figure 3.1
v	value	Figure 3.8
w	weak head-normal form	Figure 3.8
x	variable	Figure 3.1
y	variable	Figure 3.1
z	variable	Figure 3.1
A	type argument	Figure 4.1
B	module	Figure 18.1
C	constructor	Figure 3.1
D	type declaration	Figure 3.1
H	module header	Figure 18.1
I	import declaration	Figure 18.1
M	module name	Figure 18.1
P	program	Figure 3.1
R	specialization request pattern	Figure 16.2
S	type declaration shell	Figure 16.2
T	named type	Figure 3.1
X	export list entry	Figure 18.1
Υ	type constraint	Figure 6.1
$\bar{\Upsilon}$	kind constraint	Figure 16.2
α	dependency variable	Figure 6.1
β	dependency variable	Figure 6.1
γ	dependency variable	Figure 6.1
ε	empty environment/list/word	
ϑ	type tuple	Figure 9.4
κ	kind	Figure 3.1
λ	(symbol for lambda abstraction)	Figure 3.1
π	type tuple pattern	Figure 9.4
ρ	qualified kind	Figure 16.2
σ	type signature	Figure 6.1
$\bar{\sigma}$	kind signature	Figure 16.2
φ	substitution	
ψ	substitution	
Γ	type environment	Section 3.3
Δ	constraint set	Figure 6.1
	dependency environment	Section 6.4.2

$\bar{\Delta}$	kind constraint set	Figure 16.2
	kind dependency environment	Section 16.7
E	environment	
	embedding-projection pair environment	Section 11.5
Θ	type argument tuple	Figure 9.4
K	kind environment	Section 3.3
Λ	(symbol for type-level lambda abstraction)	Figure 3.1
Σ	signature environment	Section 4.4
$\bar{\Sigma}$	kind signature environment	Section 16.7
Ψ	structural representation environment	Section 11.5

Samenvatting in het Nederlands



Dit proefschrift bekijkt Generic Haskell – een uitbreiding van de functionele programmeertaal Haskell – van alle kanten. De naam "Generic Haskell" is namelijk de afgelopen jaren voor een behoorlijk aantal verschillende ideeën en talen gebruikt. Er zijn veel artikelen gepubliceerd, die allemaal een iets andere versie van de taal beschrijven. De verschillen doen zich voor in syntax, features, theorie en meer. Eén van de doelen van dit proefschrift is om de mogelijke verwarring weg te nemen. De huidige stand van zaken m.b.t. Generic Haskell wordt in zijn geheel beschreven in een consistente notatie. Dit proefschrift bevat een integrale beschrijving van Generic Haskell samen met voorgestelde uitbreidingen. De taal en de uitbreidingen worden geïntroduceerd met aanschouwelijke voorbeelden. Verder wordt beschreven hoe de componenten van de taal geïmplementeerd kunnen worden.

Van statische types naar generiek programmeren

Statische types ondersteunen het schrijven van correcte programma's. Veel programmeertalen hebben statische typering. Door gebruik te maken van statische types kunnen niet alle programmeerfouten worden gevonden, maar een goed typesysteem is in staat om een aantal, soms lastige, run-time fouten te elimineren.

Omdat besturingssystemen voor sommige van deze fouten geen goede uitzonderingsbehandelingen hebben, is het belangrijk ze al in een vroegtijdig stadium op te sporen. De fouten worden vaak veroorzaakt doordat een programma op een plek in het geheugen wil lezen of schrijven, die niet bij het programma hoort. Met behulp van statische typering kan een programma al tijdens de vertaling naar machinecode worden getest, en kan worden gecontroleerd dat bovenstaand gedrag niet voorkomt.

Typesystemen verschillen vooral in hoeveel informatie over een programma kan worden geverifieerd. Sommige typesystemen proberen bijvoorbeeld ook deling door nul te voorkomen, of dat een array index buiten de grenzen van het array ligt. Maar er is een trade-off: als het typesysteem te slim wil zijn, wordt de typechecking procedure heel inefficiënt of zelfs onbeslisbaar. En als een typesysteem weinig intelligentie bevat, dan kan een vertaler programma's afwijzen die eigenlijk goedgekeurd zouden moeten worden. De vertaler kan dan bijvoorbeeld niet herkennen dat een gevaarlijke constructie alleen in omstandigheden die aan bepaalde condities voldoen wordt gebruikt.

Het typesysteem van Hindley (1969) en Milner (1978) kan de meeste gebruikelijke programma's typeren. Het leidt op een efficiënte manier het best mogelijke type voor een programma af, zonder dat een programmeur het programma hoeft te annoteren. Een verder pluspunt van dit systeem is de mogelijkheid om *parametrisch polymorfe functies* te kunnen schrijven. Een parametrisch polymorfe functie werkt op dezelfde manier voor alle datatypes, en hoeft maar één keer geschreven te worden. Vervolgens kan zo'n functie dan automatisch voor alle datatypes geïnstantieerd worden.

Haskell (Peyton Jones 2003), maar ook andere programmeertalen zoals bijvoorbeeld SML (Milner *et al.* 1997) en Clean (Plasmeijer and van Eekelen 2001), zijn gebaseerd op het Hindley-Milner typesysteem. Vooral Haskell wordt veel gebruikt bij experimenten met uitbreidingen van het Hindley-Milner typesysteem. De definitie van Haskell bevat al een aantal uitbreidingen van het klassieke Hindley-Milner systeem. Het bevat bijvoorbeeld expliciete typesignaturen, wardoor een type van een functie kan worden ingeperkt, en met behulp van type klassen kunnen overloaded functies worden geschreven. Het gedrag van een overloaded functie hangt af van het type waarop het wordt gebruikt.

Haskell en de andere talen gebaseerd op het Hindley-Milner typesysteem hebben echter een probleem: er is een conflict tussen het gebruik van statische types om typefouten te voorkomen en het doel zo veel mogelijk code te hergebruiken. Statische types introduceren voor de computer een verschil waar er normaal geen verschil zou zijn. Bijvoorbeeld, een datum bestaat uit drie integers, maar als een datum wordt opgevat als een apart datatype, dan kunnen alleen speciale functies, die voor waardes van het type datum bedoelt zijn, toegepast worden.

Hoewel dit gedrag vaak gewenst is, kan het soms ook in de weg staan. Parametrisch polymorfe functies helpen hier. Het werken met veel verschillende datatypes wordt vereenvoudigd omdat er veel functies zijn die op alle datatypes werken. Voorbeelden van parametrisch polymorfe functies zijn: het plaatsen van een element in een lijst of een boom, of het selecteren of herschikken van elementen in een datastructuur.

Maar met parametrisch polymorfe functies kan men alleen maar dingen doen die volledig onafhankelijk zijn van de eigenlijke waardes. Vaak wil men echter de *structuur* van een datatype gebruiken in de definitie van een functie. Stel dat er drie types zijn voor identificatienummers, kamernummers en jaartallen. Al deze types zijn in principe integers, maar men wil graag voorkomen dat een jaar opeens als een kamernummer wordt gebruikt. Niettemin is het soms beter om direct met de onderliggende integers te werken, omdat er operaties voor integers bestaan die niet op alle datatypes werken. Integers kunnen worden gesommeerd, vergeleken, opgehoogd enzovoorts. Deze functionaliteit kan niet worden uitgedrukt via een parametrisch polymorfe functie, omdat het alleen toepasbaar is op integers en misschien andere numerieke types, maar niet op alle types, en zeker niet op alle types op dezelfde manier.

Met een type klasse kan een programmeur een overloaded versie van een functie schrijven, die op verschillende datatypes op een verschillende manier werkt. Een voorbeeld van een overloaded functie is de in Haskell voorgedefinieerde gelijkheidstest. Andere voorbeelden zijn optellen en vergelijken. Als identificatienummers, kamernummers en jaartallen als een instantie van de juiste type klassen worden gedeclareerd, dan kunnen we dezelfde functies voor alle drie types gebruiken. Maar wanneer we een nieuw datatype definiëren, dan moeten we ook nieuwe instance declaraties voor deze type klassen verzinnen, en als we een nieuwe functie definiëren die in principe op integers werkt, moeten we de functie zelf aan een type klasse toevoegen en alle instance declaraties aanpassen.

Er ontbreekt dus een gecontroleerd mechanisme om de verschillen die door het typesysteem worden gemaakt te negeren, en alleen maar naar de structuur van de types te kijken. Het definiëren van functies met behulp van analyse van de structuur van datatypes is waar *generiek programmeren* om draait.

Generic Haskell

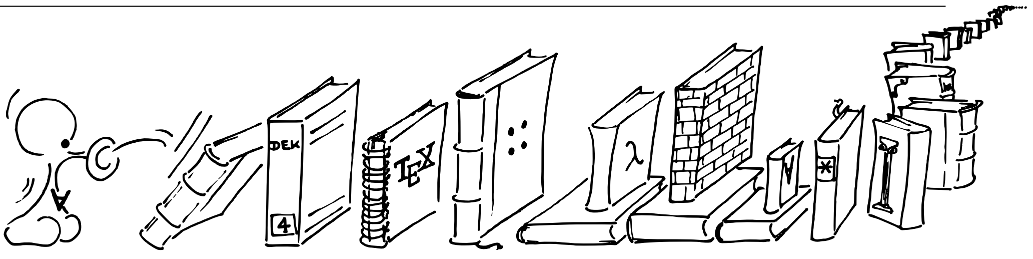
Haskell heeft al een beperkt mechanisme voor de gewenste functionaliteit: met behulp van het **deriving** construct kunnen voor een vast aantal type klassen automatisch instanties worden gegenereerd, waaronder gelijkheid, het vergelijken van waardes, het opleveren van een kleinste en grootste waarde van een type, of het vertalen van een waarde naar een canonieke string-representatie. Haskell heeft dus een aantal ingebouwde generieke programma's, maar geen taalconstructies waarmee men zijn eigen generieke programma's kan schrijven. Als men een andere generieke functie of een variant van een voorgedefinieerde functie nodig heeft, moet deze functie voor alle datatypes opnieuw gedefinieerd worden.

In de afgelopen jaren zijn er meerdere talen voor generiek programmeren ontwikkelt. Eén daarvan is Generic Haskell. Generic Haskell breidt Haskell uit met een constructie waarmee generieke functies en zelfs generieke datatypes kunnen worden gedefinieerd. Deze worden voor een beperkt aantal eenvoudige datatypes gedefinieerd, maar kunnen dan op alle (of bijna alle) datatypes in Haskell gebruikt worden. Als een nieuw datatype wordt gedefinieerd of geïmporteerd, dan zijn generieke functies automatisch ook voor dit nieuwe datatype beschikbaar.

In de context van Generic Haskell zijn er twee verschillende manieren ontwikkeld om generieke functies te definiëren. Deze zijn allebei gebaseerd op artikelen van Ralf Hinze. De eerste manier (Hinze 2000*b*) is makkelijker te gebruiken. De tweede (Hinze 2000*c*) is krachtiger, maar vereist meer kennis over de theorie van generiek programmeren. Eén van de belangrijkste bijdragen van dit proefschrift is een combinatie van deze twee stijlen, genaamd "Dependency-style" (Löh *et al.* 2003).

Wij gebruiken "Dependency-style" niet alleen om generiek programmeren in Haskell uit te leggen, maar ook om alle varianten en verbeteringen van de taal, zoals generieke abstracties, "default cases", en type-indexed datatypes te introduceren.

Bibliography



P. Achten, A. Alimarine, and R. Plasmeijer. *When generic functions use dynamic values*. In: R. Peña and T. Arts (editors), *Implementation of Functional Languages: 14th International Workshop, IFL 2002, Madrid, Spain, September 16–18, 2002, Revised Selected Papers*, volume 2670 of *Lecture Notes in Computer Science*, pages 17–33. Springer-Verlag 2003.

P. Achten, M. van Eekelen, and R. Plasmeijer. *Generic graphical user interfaces*. In: *Implementation of Functional Languages: 15th International Workshop, IFL 2003, Edinburg, Scotland, September 8–10, 2003, Revised Selected Papers*. Lecture Notes in Computer Science, Springer-Verlag 2004. To appear.

P. Achten and R. Hinze. *Combining generics and dynamics*. Technical Report NIII-R0206, Nijmegen Institute for Computing and Information Sciences, Faculty of Science, University of Nijmegen 2002.

A. Alimarine and R. Plasmeijer. *A generic programming extension for Clean*. In: T. Arts and M. Mohnen (editors), *Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers, Älvsjö, Sweden*, volume 2312 of *Lecture Notes in Computer Science*, pages 168–185. Springer-Verlag 2001.

A. Alimarine and S. Smetsers. *Optimizing generic functions*. In: *Proceedings of the Seventh International Conference on Mathematics of Program Construction, Stirling, United Kingdom* 2004. To appear.

T. Altenkirch and C. McBride. *Generic programming within dependently typed programming*. In: J. Gibbons and J. Jeuring (editors), *Generic Programming: IFIP TC2/WG2.1 Working Conference on Generic Programming July 11–12, 2002, Dagstuhl, Germany*, pages 1–20. Number 115 in International Federation for Information Processing, Kluwer Academic Publishers 2003.

F. Atanassow and J. Jeuring. *Inferring type isomorphisms generically*. In: *Proceedings of the Seventh International Conference on Mathematics of Program Construction, Stirling, United Kingdom* 2004. To appear.

L. Augustsson. *Cayenne – a language with dependent types*. ACM SIGPLAN Notices, 34(1):pages 239–250 1999.

A. I. Baars and S. D. Swierstra. *Typing dynamic typing*. In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166. ACM Press 2002.

R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. *Generic programming: An introduction*. In: S. D. Swierstra, P. R. Henriques, and J. N. Oliveira (editors), *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115. Springer-Verlag 1999.

R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Europe, London, second edition 1998.

R. Bird, O. de Moor, and P. Hoogendijk. *Generic functional programming with types and relations*. Journal of Functional Programming, 6(1):pages 1–28 1996.

M. M. T. Chakravarty and G. Keller. *An approach to fast arrays in Haskell*. In: J. Jeuring and S. Peyton Jones (editors), *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures*, volume 2638 of *Lecture Notes in Computer Science*, pages 27–58. Springer-Verlag 2003.

J. Cheney and R. Hinze. *A lightweight implementation of generics and dynamics*. In: *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 90–104. ACM Press 2002.

J. Cheney and R. Hinze. *First-class phantom types*. Technical Report CUCIS TR2003-1901, Cornell University 2003.

URL `http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2003-1901/`

K. Claessen and J. Hughes. *QuickCheck: A lightweight tool for random testing of Haskell programs*. In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279. ACM Press 2000.

D. Clarke, R. Hinze, J. Jeuring, A. Löh, and J. de Wit. *The Generic Haskell user's guide, version 0.99 (Amber)*. Technical Report UU-CS-2001-26, Institute of Information and Computing Sciences, Universiteit Utrecht 2001.

D. Clarke, J. Jeuring, and A. Löh. *The Generic Haskell user's guide, version 1.23 – Beryl release*. Technical Report UU-CS-2002-047, Institute of Information and Computing Sciences, Universiteit Utrecht 2002.

D. Clarke and A. Löh. *Generic Haskell, specifically*. In: J. Gibbons and J. Jeuring (editors), *Generic Programming: IFIP TC2/WG2.1 Working Conference on Generic Programming July 11–12, 2002, Dagstuhl, Germany*, pages 21–47. Number 115 in International Federation for Information Processing, Kluwer Academic Publishers 2003.

L. Damas and R. Milner. *Principal type-schemes for functional programs*. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 207–212. ACM Press 1982.

GHC Team. *The Glasgow Haskell Compiler User's Guide*.
URL `http://haskell.org/ghc/docs/latest/users_guide.ps.gz`

J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Ph.D. thesis, Université Paris VII 1972.

R. Harper and G. Morrisett. *Compiling polymorphism using intensional type analysis*. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 130–141. ACM Press 1995.

R. Hindley. *The principal type-scheme of an object in combinatory logic*. Transactions of the American Mathematical Society, 146:pages 29–66 1969.

R. Hinze. *A generic programming extension for Haskell*. In: E. Meijer (editor), *Proceedings of the Third Haskell Workshop, Paris, France*. Number UU-CS-1999-28 in Universiteit Utrecht Technical Reports 1999*a*.

R. Hinze. *Polytypic functions over nested datatypes*. Discrete Mathematics and Theoretical Computer Science, 3(4):pages 193–214 1999*b*.

R. Hinze. *Generic Programs and Proofs* 2000*a*. Habilitationsschrift, Bonn University.

R. Hinze. *A new approach to generic functional programming*. In: T. W. Reps (editor), *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts*, pages 119–132. ACM Press 2000*b*.

R. Hinze. *Polytypic values possess polykinded types*. In: R. Backhouse and J. N. Oliveira (editors), *Proceedings of the Fifth International Conference on Mathematics of Program Construction, July 3–5, 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer-Verlag 2000*c*.

R. Hinze. *Fun with phantom types*. In: J. Gibbons and O. de Moor (editors), *The Fun of Programming*. Cornerstones of Computing, Palgrave Macmillan 2003.

R. Hinze and J. Jeuring. *Generic Haskell: applications*. In: R. Backhouse and J. Gibbons (editors), *Generic Programming, Advanced Lectures*, volume 2793 of *Lecture Notes in Computer Science*, pages 57–96. Springer-Verlag 2003*a*.

R. Hinze and J. Jeuring. *Generic Haskell: practice and theory*. In: R. Backhouse and J. Gibbons (editors), *Generic Programming, Advanced Lectures*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer-Verlag 2003*b*.

R. Hinze, J. Jeuring, and A. Löh. *Type-indexed data types*. In: E. A. Boiten and B. Möller (editors), *Mathematics of Program Construction: Sixth International Conference*, volume 2386 of *Lecture Notes in Computer Science*, pages 148–174. Springer 2002. Also appeared as Universiteit Utrecht Technical Report UU-CS-2002-011.
URL `http://www.cs.ukc.ac.uk/pubs/2002/1368/`

R. Hinze and S. Peyton Jones. *Derivable type classes*. In: G. Hutton (editor), *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41(1) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science 2001. The preliminary proceedings appeared as a University of Nottingham technical report.

G. Huet. *Functional Pearl: The Zipper*. Journal of Functional Programming, 7(5):pages 549–554 1997.

P. Jansson. *Functional Polytypic Programming*. Ph.D. thesis, Chalmers University of Technology and Göteborg University 2000.

P. Jansson and J. Jeuring. *PolyP – a polytypic programming language extension*. In: *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France*, pages 470–482. ACM Press 1997.

C. B. Jay. *Distinguishing data structures and functions: the constructor calculus and functorial types*. In: S. Abramsky (editor), *Typed Lambda Calculi and Applications: 5th International Conference TLCA 2001, Kraków, Poland, May 2001 Proceedings*, volume 2044 of *Lecture Notes in Computer Science*, pages 217–239. Springer-Verlag 2001.

C. B. Jay. *The pattern calculus* 2003. Accepted for publication by ACM Transactions on Programming Languages and Systems.
URL `http://www-staff.it.uts.edu.au/~cbj/Publications/pattern_calculus.ps`

C. B. Jay, E. Moggi, and G. Bellè. *Functors, types and shapes*. In: R. Backhouse and T. Sheard (editors), *Workshop on Generic Programming: Marstrand, Sweden, 18th June, 1998*. Chalmers University of Technology 1998.

M. P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press 1994.

M. P. Jones. *Type classes with functional dependencies*. In: *Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244. Springer-Verlag 2000.

W. Kahl and J. Scheffczyk. *Named instances for Haskell type classes*. In: R. Hinze (editor), *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*. Number UU-CS-2001-62 in Technical Report, Institute of Information and Computing Sciences, Universiteit Utrecht 2001.

P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. *GAST: Generic Automated Software Testing*. In: R. Peña and T. Arts (editors), *Implementation of Functional Languages: 14th International Workshop, IFL 2002, Madrid, Spain, September 16–18, 2002, Revised Selected Papers*, volume 2670 of *Lecture Notes in Computer Science*, pages 84–100. Springer-Verlag 2003.

R. Lämmel and S. Peyton Jones. *Scrap your boilerplate: a practical design pattern for generic programming*. ACM SIGPLAN Notices, 38(3):pages 26–37 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

R. Lämmel and S. Peyton Jones. *Scrap more boilerplate: reflection, zips, and generalised casts* 2004. Draft; submitted to ICFP 2004.

R. Lämmel and J. Visser. *Typed combinators for generic traversal*. In: *Proceedings Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *Lecture Notes in Computer Science*, pages 137–154. Springer-Verlag 2002.

R. Lämmel, J. Visser, and J. Kort. *Dealing with large bananas*. In: J. Jeuring (editor), *Workshop on Generic Programming 2000, Ponte de Lima, Portugal*, volume UU-CS-2000-19 of *Universiteit Utrecht Technical Report*, pages 46–59 2000.

J. R. Lewis, M. B. Shields, E. Meijer, and J. Launchbury. *Implicit parameters: Dynamic scoping with static types*. In: T. W. Reps (editor), *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts*, pages 108–118 2000.

A. Löh, D. Clarke, and J. Jeuring. *Dependency-style Generic Haskell*. In: *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 141–152. ACM Press 2003.

I. Lynagh. *Typing Template Haskell: Soft types* 2004. Submitted to Haskell Workshop 2004.

C. McBride. *The derivative of a regular type is its type of one-hole contexts* 2001. Unpublished manuscript.

C. McBride and J. McKinna. *The view from the left*. Journal of Functional Programming, 14(1):pages 69–111 2004.

R. Milner. *A theory of type polymorphism in programming*. Journal of Computer and System Sciences, 17(3):pages 348–375 1978.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press 1997.

U. Norell and P. Jansson. *Polytypic programming in Haskell*. In: *Implementation of Functional Languages: 15th International Workshop, IFL 2003, Edinburg, Scotland, September 8–10, 2003, Revised Selected Papers*. Lecture Notes in Computer Science, Springer-Verlag 2004. To appear.

S. Peyton Jones (editor). *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press 2003.

S. Peyton Jones and M. Shields. *Practical type inference for arbitrary-rank types* 2003. URL `http://research.microsoft.com/Users/simonpj/papers/putting/`

B. C. Pierce. *Types and Programming Languages*. The MIT Press 2002.

R. Plasmeijer and M. van Eekelen. *The Concurrent Clean language report, version 2.0* 2001.
URL `ftp://ftp.cs.kun.nl/pub/Clean/Clean20/doc/CleanRep2.0.pdf`

T. Sheard and S. Peyton Jones. *Template meta-programming for Haskell*. ACM SIGPLAN Notices, 37(12):pages 60–75 2002.

M. G. T. Van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. *Efficient annotated terms*. Software – Practice and Experience, 30(3):pages 259–291 2000.

M. de Vries. *Specializing Type-Indexed Values by Partial Evaluation*. Master's thesis, Rijksuniversiteit Groningen 2004.

D. Vytiniotis, G. Washburn, and S. Weirich. *An open and shut typecase* 2004. Submitted to ICFP 2004.

W3C. *XML Schema: Formal description* 2001.
URL `http://www.w3.org/TR/xmlschema-formal/`

W3C. *Extensible markup language (XML) 1.0 (third edition), W3C recommendation* 2004.
URL `http://www.w3.org/TR/2004/REC-xml-20040204/`

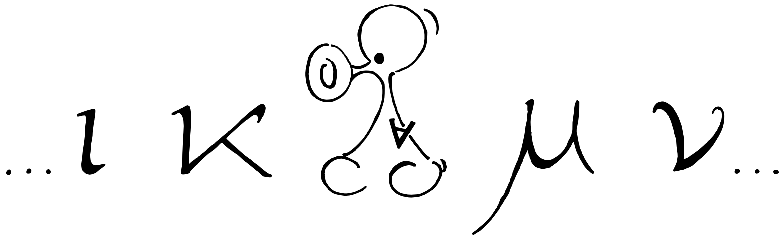
P. Wadler. *Views: a way for pattern matching to cohabit with data abstraction*. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313. ACM Press 1987.

P. Wadler and S. Blott. *How to make ad-hoc polymorphism less ad hoc*. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM Press 1989.

P. L. Wadler. *How to replace failure by a list of successes*. In: J. P. Jouannaud (editor), *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128. Springer-Verlag 1985.

S. Weirich. *Higher-order intensional type analysis*. In: D. L. Métayer (editor), *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002*, volume 2305 of *Lecture Notes in Computer Science*, pages 98–114. Springer-Verlag 2002.

Index



Curriculum vitae

Andres Löh

18 August 1976	born in Lübeck, Germany
1986 – 1995	grammar school, Katharineum zu Lübeck, Germany
10 June 1995	school-leaving exam
1995 – 2000	studies of Mathematics with Computer Science as subsidiary subject at the University of Konstanz, Germany
1996 – 2000	scholarship granted by the German National Scholarship Foundation
17 August 2000	"Diplom"
2000 – 2004	"assistent in opleiding" at Utrecht University, Netherlands

Titles in the IPA Dissertation Series

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Schedulere Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Progam Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11