Formal Polytypic Programs and Proofs

Wilhelmina Johanna Verbruggen

February 1, 2010

Declaration

Wilhelmina Johanna Verbruggen

Summary

Polytypic or datatype-generic programming is a form of generic programming where we abstract over the shape of datatypes: we define functions by induction over the structure of datatypes. In this thesis we show how to formalize polytypic programming within the proof assistant Coq and show how we can do fully formal proofs over polytypic programs.

Although the formalization of polytypic functions and polytypic proofs is challenging, we provide an easy to use interface for the user which allows to define polytypic functions in a way that is familiar from Generic Haskell and makes it possible to give straight-forward proofs of properties of polytypic functions. Formal proofs are often time consuming, but support for polytypic proofs reduces the so-called "burden of proof". A single polytypic proof corresponds to a family of proofs, proving that the property holds for each instance of the polytypic function.

To define a polytypic function, a programmer must give instances for each type constant (unit, int, sum and product) and give the type of the function when specialized to types of kind $\star$. Given these, the function can be specialized to any datatype of any kind. We have reified the notion of a polytypic function within Coq as a datatype `PolyFn` with fields for exactly these components, and programmers familiar with Generic Haskell should have little difficulty in translating their Generic Haskell definitions to Coq. Function specialization is then defined as an ordinary (dependently typed) function within Coq. It takes a `PolyFn` as input and returns something of the type computed by type specialization. This can be regarded as a formal proof that term specialization is correct with respect to type specialization. The main difficulties in this part of the development were the need for impredicativity of the type associated with kind $\star$ and the absence of universe polymorphism.

Implementing polytypic programming within a dependently typed language (such as Coq) gives us some distinct advantages. Since we can formalize type specialization within the host language, we get type checking of polytypic functions virtually for free. Furthermore, since polytypic functions are represented by an ordinary datatype within the host language they are first-class citizens. We can define polytypic functions in terms of other polytypic functions and it becomes possible to define combinators on polytypic functions.

We extend the development with polytypic properties and proofs. The structure of a polytypic proof follows the structure of a polytypic function closely: the user gives

a proof for each of the type constants and gives the property for types of kind $\star$. Like for functions, we provide a lemma that says that polytypic proofs can be specialized to arbitrary datatypes of arbitrary kinds. This can be interpreted as a formal proof that to prove a property of a polytypic function it is indeed sufficient to give the instances of the proof for the type constants.

Most problems in this part of the development stemmed from the fact that in the specialization of a polytypic proof we need to refer to the specialization of polytypic functions. A proof specialized to type T is a proof that a property holds for a particular polytypic function *specialized to* T. Since the definition of term specialization involves a number of type conversions, reasoning about these specialized terms involves reasoning about heterogeneous equalities. However, while our definition of proof specialization is rather involved this does not affect proofs as seen by the user. The user still only needs to provide the proofs for the type constants. These proofs are straight-forward, and it should be possible to automate them to some extent through the use of a custom tactics library.

We discuss how we can deal with (co)inductive datatypes and proofs, following the same approach that is taken in Generic Haskell. Unfortunately, this requires some manual work on behalf of the programmer or the prover, and the approach is limited because of restrictions posed by the Coq guardedness checker. Nevertheless, our proof of concept demonstrates that the Generic Haskell approach is feasible in a formal setting. Making the approach more generic, or lifting the restrictions of the guardedness checker, is discussed in future work.

We go on to review work related to our own. We mention a number of approaches to generic programming and explain the difference to the approach on which this thesis is based. We then compare our development to a number of other implementations of polytypic programming in dependently typed languages, and discuss some work that is of interest in improving our implementation of recursion.

We can extend the work in this thesis in a number of ways. There is some scope for automation and syntactic sugar to make defining polytypic functions and proofs more intuitive. We would like to extend the definition of polytypic properties to allow for properties about multiple polytypic functions. Other useful extensions include type-indexed types, the extraction of code to Generic Haskell and the addition of meta-information to our type universe to allow for such polytypic functions as pretty printers and parsers.

We conclude that our infrastructure allows for Generic Haskell-style polytypic programming in the proof assistant Coq and that we have shown that fully formal (polytypic) proofs about these programs can be straight-forward.

Acknowledgements

First and foremost I want to thank Dr Edsko de Vries for getting me through the past four years. He has been a tremendous help, both on a technical and personal front, and without him this thesis would not exist.

I thank my supervisor Arthur Hughes for his input in this work, for taking the time to listen to me trying to explain things over a cup of coffee, and supporting me in any way possible.

Thanks is also due to everyone who provided me with helpful feedback during the course of this work. Most recently, the people attending the Workshop on Generic Programming 2009 were very supportive and I am grateful to Stephanie Weirich, Conor McBride, Patrik Jansson and Philip Wadler for their helpful comments and suggestions.

I am thankful to Ralf Hinze, Johan Jeuring and Yves Bertot for early feedback on my research proposal, and to Andrew Butterfield and William Harrison for their comments during the transfer process.

My colleagues in the Foundations and Methods Group have been supportive and through weekly meetings have introduced me to various other interesting aspects of theoretical computer science. A thank-you also to my office mates, especially Daniel Isemann and David Eadie, for many random conversations about nothing in particular over much needed cups of tea.

This thesis is based on two papers published at the Workshop on Generic Programming: (Verbruggen et al., 2008, 2009), and an abridged version of this thesis will appear in a special issue on Generic Programming of the Journal of Functional Programming (Verbruggen et al., 2010). The work done in these papers is all my own work, with insightful advice and help on some very particular problems in the development given by Edsko de Vries and Arthur Hughes, who appear as co-authors on the papers.

This work has been supported by the Irish Research Council for Science, Engineering and Technology (IRCSET).

Contents

List of Figures

Chapter 1

Introduction

The endless quest for higher levels of abstractions is one of the most fundamental topics in programming language research and in computer science in general. Abstraction is absolutely essential to computer science; even the most primitive programming languages depend on abstract concepts such as bytes or variables: abstractions over sequences of bits (an abstraction because they can be ordered high to low or low to high) and memory locations (an abstraction because the programmer does not have to worry about the exact value of the address, or deal with machine hardware details such as segmentation). As the complexity of computer systems increases, abstraction needs to increase accordingly: from simple concepts such as bytes to "larger" concepts such as procedures, abstract datatypes, application programmer interfaces or network protocols, to advanced concepts such as encapsulation in modern day imperative languages or mathematically inspired abstractions such as monads in functional languages.

Most sciences, including mathematics itself, attempt to build abstract mathematical models of the domain of discourse in order to study its properties. Abstraction here is key as it allows to ignore inessentials: Pythagoras' theorem about the length of the sides of a triangle does not depend on the material that was used to build the triangle. This kind of abstraction can be characterized as *information neglect* (Colburn and Shute, 2007). Abstraction as used in computer science is different in a subtle but important way, and can be described as *information hiding*: information that is essential in one context (the exact ordering of the bits in a byte) can be ignored in another context. It is this distinction that makes it possible to develop large software systems.

Abstraction is correspondingly important when we move beyond merely writing software to mathematically proving that our software satisfies a given specification. Software written at a low level of abstraction will be difficult to prove correct as there are many irrelevant details that need to be taken into account. When we raise the level of abstraction these details disappear and the proofs will be simpler. Again, this is not just a case of information neglect but of information hiding: if a programming language deals with variables rather than naked memory locations we can assume that all variables correspond to valid areas of memory. In fact, the formal model used for reasoning would not need to deal with "real" memory addresses at all. In contrast, languages such as C which break this abstraction and allow address arithmetic on variables are notoriously difficult to reason about.

When we do proofs about software systems, the use of abstraction is not only important in the construction of the software but also in the construction of the proof itself. This should come as no surprise: one of the most fundamental results of computer science is the Curry-Howard isomorphism which (informally) states that proofs *are* programs. Consequently, many of the concerns of software engineering also apply to proof engineering. This is especially important when we want to construct formal—that is, machine verified—proofs. Information neglect is commonplace in informal mathematical proofs where "inessential" details are simply left out, but is not usable when constructing proofs to be mechanically verified. Instead, we need to make use of information hiding.

Genericity is almost as general a term as abstraction; it refers to parametrization over some aspect of a system. The kind of genericity that we are interested in here is parametrization over a datatype. This can take various forms. In (qualified) parametric polymorphism (in functional languages) or templates (in imperative languages) parts of the datatype are completely hidden and as a consequence programs must operate uniformly over elements of these datatypes. The kind of genericity that is the topic of this thesis is often referred to as *datatype-genericity* or *polytypicity*: the program can abstract over a datatype but can nevertheless take advantage of the *structure* of the datatype. As we shall see, this is a powerful abstraction mechanism which can be used both to define programs *and* to define proofs over these programs.

In Section 1.1 we will discuss the topic of genericity in programming languages in more detail, focusing eventually on polytypic programming. The topic of this thesis is the construction of *formal* proofs over polytypic programs. We motivate the need for formal proofs in Section 1.2 and briefly discuss the proof environment that we will use. Finally, we will give an overview of the rest of the thesis in Section 1.4.

1.1 Generic Programming

Generic programming is a much over-used term, and to different people has different meanings. What they all have in common is that they provide parametrization over some aspect of a programming language, making it possible to write more general programs.

1.1.1 Parametrization by Type

Parametric polymorphism is a well-known example of such an abstraction, in this case abstraction over a type. Let us take lists as an example. In Haskell, we can define two separate datatypes to represent lists of integers and lists of characters:

```
data ListInt = INil | ICons Int ListInt
data ListChar = CNil | CCons Char ListChar
```

These two datatypes are identical except for the type of the elements contained in the list. We can combine them into a single parametric datatype[1] `List`, which takes a type a as its argument and constructs lists containing elements of type a:

```
data List a = Nil | Cons a (List a)
```

To get the type of integer lists, all we need to do is instantiate a by the type `Int`. Given a parametric type we can write a polymorphic function, which abstracts over the parameters of the type. A function that takes an element of type `List` a as input does not have any information about the type of the elements contained in the list: it cannot examine the elements of the list, it can only rearrange or count them. Two examples of polymorphic functions on lists are the `length` and `append` functions:

```
length :: List a → Int
length Nil = 0
length (Cons x xs) = 1 + length xs

append :: List a → List a → List a
append Nil l = l
append (Cons x xs) l = Cons x (append xs l)
```

1.1.2 Parametrization by Function

Another form of genericity is parametrization by functions. This approach is a good fit for functional languages, because they treat functions as first-class citizens. Functions parametrized by other functions are called *higher-order functions*. An example of a higher-order function is the mapping function for lists, which abstracts over the function f that is mapped across the list:

```
mapList :: (a → b) → List a → List b
mapList f Nil = Nil
mapList f (Cons x xs) = Cons (f x) (mapList f xs)
```

1.1.3 STL and Type Classes

In the world of object-oriented programming, generic programming is probably most often associated with the Standard Template Library (STL) in C++. This library provides containers, iterators and algorithms for many datatypes, and therefore allows for many different forms of abstraction. There is a similar notion in Haskell called *type classes*, which allows the programmer to define a function over elements of any type a as long as a has certain properties. For example, we could define a sort function on lists:

[1]The term *polymorphic datatype* is also often used to describe `List` (Gibbons, 2006, Section 2.2), but this is not strictly correct: a polymorphic type would have the form $\forall a\ .\ t$, whereas the type of `List` is $\lambda a\ .\ t$.

```
sort :: Ord a => List a → List a
```

which acts on lists containing elements of any type *a*, as long as *a* satisfies the properties defined by the type class `Ord` (in this case, the elements of type *a* must be comparable).

There are many more examples of different types of generic programming out there, but discussing them all is beyond the scope of this thesis. We refer the reader to (Gibbons, 2006) for a comprehensive classification of generic programming methods. For a thorough survey and comparison of generic programming methods used by the functional programming community, see (Hinze et al., 2006a). We will now zoom in on the branch of generic programming that we will use in this thesis.

1.1.4 Polytypic Programming

The idea of polytypic programming first appeared in the language *PolyP* (Jansson and Jeuring, 1997), and a number of related approaches have appeared since such as the style of polytypic programming often referred to as *origami programming* (Gibbons, 2006), and the approach of kind-indexed types (Hinze, 2000b) as implemented in Generic Haskell. This last approach is the one that is the topic of this thesis.

The core concept of polytypic programming is that functions are defined by induction on the *structure* of datatypes. Where for parametric polymorphism we abstract from the occurrence of `Int` in `List Int`, here we abstract form the type constructor `List` itself. The most important difference between polytypic programming and type classes is that in addition to abstracting over the structure or shape of a datatype, polytypic programming also allows us to exploit this structure when defining a function. This is in fact the main feature of polytypic programming: we can write functions that behave differently for different datatypes.

As an example, consider the equality function that can compare any two terms, as long as they have the same type (independent of what that type is). This is a type-indexed function: it will look at the type parameter it is given and based on that it will determine how to compare the two elements it gets as input; comparing two boolean values is done differently than comparing two lists. One way to implement this function is to supply definitions for all datatypes we want to apply the function to:

```
data Bool = False | True

eq⟨Bool⟩ False False = True
eq⟨Bool⟩ False True = False
eq⟨Bool⟩ True False = False
eq⟨Bool⟩ True True = True

data List a = Nil | Cons a (List a)

eq⟨List a⟩ Nil Nil = True
eq⟨List a⟩ Nil (Cons x xs) = False
```

```
eq⟨List a⟩ (Cons x xs) Nil = False
eq⟨List a⟩ (Cons x xs) (Cons y ys) =
 eq⟨a⟩ x y && eq⟨List a⟩ xs ys

data Tree a b = Leaf a | Node b (Tree a b) (Tree a b)

eq⟨Tree a b⟩ (Leaf x) (Leaf y) = eq⟨a⟩ x y
eq⟨Tree a b⟩ (Leaf x) (Node y t1 t2) = False
eq⟨Tree a b⟩ (Node x t1 t2) (Leaf y) = False
eq⟨Tree a b⟩ (Node x t1 t2) (Node y u1 u2) =
 eq⟨b⟩ x y && eq⟨Tree a b⟩ t1 u1 && eq⟨Tree a b⟩ t2 u2
```

If we would continue this for all possible datatypes we would indeed get a function that is able to compare any two elements of any type. However, this might be a little difficult because there are infinitely many possible datatypes. This is the way equality is implemented in (non-generic) Haskell, using type classes. Such an approach is often referred to as *ad-hoc polymorphism* or *overloading*. There are some obvious disadvantages to this approach. We still have to define the equality function for every datatype, and when we define a new datatype we need to expand the equality function to include it.

So let us have a look at what the equality function actually does, and try to find some similarities that we can use to make this function a little easier to write generically.

The first thing we notice is that when the constructors of the two terms we want to compare are different the terms are never equal, so in that case we can always return `False`. When the constructors are the same we want to compare the arguments to the constructors: if it has no arguments (like `Nil`) we are done and the terms are equal; otherwise we want to compare each of the arguments using the equality function recursively with the type argument set to the type of the constructor argument.

Notice that we have described the behaviour of the equality function *without using any datatype-specific information*. We can use this information to give a simple structural representation of datatypes:

- We have a number of base types, like `Int` and `Char`.
- Every other datatype consists of a series of constructors, one of which is selected each time we construct a term of that type. We represent this choice of constructors as a disjoint union, denoted by the symbol $+$.
- Each individual constructor is represented as a tuple, containing an element for each argument to the constructor. We denote this tuple of arguments by the symbol $\times$.
- If a constructor has no arguments it is represented by the unit element, denoted by `Unit`.

Hopefully some examples will clarify this. We denote the structural representation of a datatype by adding a $^\circ$ to the name:

```
data Bool = False | True
type Bool° = Unit + Unit

data List a = Nil | Cons a (List a)
type List° a = Unit + a × (List a)

data Tree a b = Leaf a | Node b (Tree a b) (Tree a b)
type Tree° a b = a + b × (Tree a b) × (Tree a b)
```

There is a technicality here: in most functional languages there is no recursion on the type level, except through algebraic datatypes. This means that we cannot refer to `List°` in the definition of `List°`, and so we must refer to `List` instead. A detailed discussion of this is beyond the scope of the current section and we will come back to it in Chapter 4.

We also need a way to build elements of these structural types, and we can do this by the following datatype definitions:

```
data Unit = Unit
data a + b = Inl a | Inr b
data a × b = (a, b)
```

This means that we must translate all elements of arbitrary types to elements that only use these structural types. For example, we should apply the following translations:

```
False :: Bool ↦
 Inl Unit :: Unit + Unit
Cons 2 (Cons 3 Nil) :: List Int ↦
 Inr (2, Inr (3, Inl Unit)) :: Unit + Int x (List Int)
```

Fortunately, this is a fairly straightforward translation which can usually be automated.

Now that we have determined the structure of datatypes, we can define a polytypic function by induction on this structure, i.e. we define the function for all base types, the unit type, the disjoint union of two types and a pair of types. We can then apply the function to any datatype that is built up of these components. We call this the *specialization* of a polytypic function to a particular type. The polytypic definition of the equality function is (where we separate the structural type argument from the other arguments by enclosing it in angular brackets for emphasis):

```
eq⟨T⟩ :: T → T → Bool
eq⟨Int⟩ i1 i2 = eqInt i1 i2
eq⟨Unit⟩ u1 u2 = True
eq⟨a+b⟩ (Inl x) (Inl y) = eq⟨a⟩ x y
eq⟨a+b⟩ (Inr x) (Inr y) = eq⟨b⟩ x y
eq⟨a+b⟩ x y  = False
eq⟨a×b⟩ (x1, y1) (x2, y2) = eq⟨a⟩ x1 x2 && eq⟨b⟩ y1 y2
```

To see how this polytypic equality function can be specialized, we will give a few examples. Comparing the values `True` and `False` of type `Bool` should return `False` because these values are not equal. We know that `Bool` translates to the structural type

`Unit + Unit`, and that `True` and `False` correspond to `Inr Unit` and `Inl Unit`, respectively. Therefore, the specialization will take the following form:

```
eq⟨Bool⟩ True False
 = eq⟨Unit + Unit⟩ (Inr Unit) (Inl Unit)
 = False
```

Because `Bool` has two possible constructors, we must first check whether the two elements we want to compare were built using the same constructor. This is not the case, and therefore the elements can never be equal.

For our next example we will use the slightly more complicated datatype `List Int`. We want to compare the list containing only the integer 2 to the list containing 2 and 3. In order to keep the example readable we will only expand definitions when necessary. `Nil` will be expanded to `Inl Unit` (leftmost constructor with no arguments) and `Cons x y` will be expanded to `Inr (x, y)` (rightmost constructor with a pair of arguments).

```
eq⟨List Int⟩ (Cons 2 Nil) (Cons 2 (Cons 3 Nil))
 = eq⟨Unit + Int × (List Int)⟩ (Inr (2, Nil))
                               (Inr (2, (Cons 3 Nil)))
 = eq⟨Int × (List Int)⟩ (2, Nil) (2, (Cons 3 Nil))
 = eq⟨Int⟩ 2 2 && eq⟨List Int⟩ Nil (Cons 3 Nil))
```

We now have two different cases we need to solve, and we will solve them separately, starting with the first case. The first case compares the integer 2 to itself, and if `eqInt` is defined correctly it should return `True`:

```
eq⟨Int⟩ 2 2
   = eqInt 2 2 -- External definition of eqInt needed
   = True
```

The second case is a little more complicated. First we need to expand the definitions of `List Int`, `Cons` and `Nil`. We end up with two different constructors (as was the case in the example for `Bool`), which will always return `False`—the empty list and the list containing the integer 3 are obviously not the same.

```
eq⟨List Int⟩ Nil (Cons 3 Nil)
 = eq⟨Unit + Int × (List Int)⟩ (Inl Unit) (Inr (3, Nil))
 = False
```

Combining our two results we get `True && False` which gives an overall result of `False`: the list containing only 2 is not the same as the list containing both 2 and 3.

In addition to the equality function we can define numerous other useful functions by induction on the structure of types. Some well-known examples include `map`, `encode`/`decode`, `zip` and `fold`.

1.1.5 Kind-Indexed Polytypic Programming

At the end of the previous section we managed to give a definition of equality that can be specialized to any type of kind $\star$. In this section we will see how we can generalize this so that we can give a definition of equality that can be specialized to types of *arbitrary* kinds: not just to `Bool` or `List Int`, but also to the type constructor `List` a. This so-called kind-indexed polytypic programming is what sets the approach to polytypic programming in Generic Haskell apart from most other approaches, and it is the style that we want to formalize in this thesis.

To understand this approach, we need to understand what we mean by the kind of a type. A kind can be thought of as a type of a type:

- We use the kind $\star$ to represent nullary constructors (constructors without arguments), e.g. `Bool`, `Int`, `Char`. We often call nullary type constructors simply *types*.
- The kind $k_1 \rightarrow k_2$ represents type constructors that map type constructors of kind k_1 to type constructors of kind k_2. For example, the type constructor `List` maps the type a to the type `List` a, both of kind $\star$. Therefore, the kind of `List` is $\star \rightarrow \star$.

To determine the kind of a type constructor there are two things to take into account: the number of type arguments and their kinds. Let us consider a few examples:

```
data Bool = False | True
-- no arguments: kind ⋆

data Maybe a = Nothing | Just a
-- 1 argument a of kind ⋆: kind ⋆ → ⋆

data Tree a b = Leaf a | Node b (Tree a b) (Tree a b)
-- 2 arguments a and b of kind ⋆: kind ⋆ → ⋆ → ⋆

data GRose f a = GBranch a (f (GRose f a))
-- 2 arguments: f of kind ⋆ → ⋆; a of kind ⋆
-- This gives an overall kind of (⋆ → ⋆) → ⋆ → ⋆ to GRose.
```

In other words: we can view the kind $\star$ as the kind of types that actually contain values, and the kind $\star \rightarrow \star$ as the kind of type constructors that take a type of kind $\star$ and return a type of kind $\star$ (i.e. it takes a type containing values and returns a type containing values). For example, there are values of type `Bool` $:: \star$ and `Int` $:: \star$ and there are no values of type `List` $:: \star \rightarrow \star$, but there are values of type `List Bool` $:: \star$ and `List Int` $:: \star$.

We can now rewrite the signature for the polytypic equality function `eq` to

```
eq⟨T :: ⋆⟩ :: T → T → Bool
```

emphasizing that `eq` is indexed by a type T of kind $\star$. Consider using the polytypic equality function to compare two lists—we will need some way to compare the elements of the lists. If we want to index the equality function by a type of a different kind, the type of the polytypic equality function changes:

```
eq⟨T :: ⋆ → ⋆⟩ :: (a → a → Bool) → T a → T a → Bool
```

In order to compare two lists, we must provide it with a function that can compare the elements of the lists. One solution would be to write a different polytypic equality function for each possible kind. This is a significant improvement over having to write a separate function for every *type*, since the number of different kinds we might use is likely to be reasonably small. Ideally we would like a single equality function that works for all types of all kinds, but since the number of kinds is infinite we would never be able to cover them all. Even if we would write an equality function for every kind that we actually need, this goes against the generic programming philosophy. What we want is a single equality function that works for *all* types.

The problem is that we cannot give a straight-forward type for this equality function: the type changes depending on the kind of T. The key insight in (Hinze, 2000b) is that a type-indexed function must have a kind-indexed type:

```
eq⟨T :: k⟩ :: Equal⟨k⟩ T
```

where `Equal` is used as a type constructor indexed by a kind k, and can be defined as:

```
Equal⟨k :: □⟩ :: k → ⋆
Equal⟨⋆⟩ T = T → T → Bool
Equal⟨k1 → k2⟩ T = ∀a . Equal⟨k1⟩ a → Equal⟨k2⟩ (T a)
```

For the case where `T` :: ⋆, the type of the equality function will still be `T → T → Bool`, but for any other kinds the type will be different.

We can now give a type to the equality function for both $T =$ `Int` :: ⋆ and $T =$ `List` :: ⋆ → ⋆ (and all other types of any kind):

```
eq⟨Int :: ⋆⟩
  :: Equal⟨⋆⟩ Int
   = Int → Int → Bool

eq⟨List :: ⋆ → ⋆⟩
  :: Equal⟨⋆ → ⋆⟩ List
   = ∀a . Equal⟨⋆⟩ a → Equal⟨⋆⟩ (List a)
   = ∀a . (a → a → Bool) → List a → List a → Bool
```

The definition of `eq` also changes slightly. The previous definition relied on explicit recursion for the sum and product case. However, where `eq` always received two parameters before, it now receives a variable number of parameters, depending on the kind of the type argument. For T :: ⋆, it gets two parameters: the two terms we want to compare for equality; but for T :: ⋆ → ⋆ it needs three parameters: the two terms to compare (e.g. two lists) and a function to compare the elements contained in them. Since this number is variable, it is impossible to explicitly list these parameters as arguments to the polytypic function, or indeed use them in the definition when we recursively call `eq`. To solve this problem, the recursion is abstracted out, and the sum and product case receive two additional parameters `eqA` and `eqB` that can be used to recursively determine equality on terms of type a and b.

We can also think about this in a different way: rather than giving an instance for $a + b$ and $a \times b$ for some specific a and b—that is, for types of kind $\star$—we give an instance for $(+)$ and for $(\times)$—types of kind $\star \to \star \to \star$. Hence, the instance for these two cases will be provided with two additional arguments.

```
eq⟨T :: k⟩ :: Equal⟨k⟩ T
eq⟨Int⟩ i1 i2 = eqInt i1 i2
eq⟨Unit⟩ u1 u2 = True
eq⟨+⟩ eqA eqB (Inl x) (Inl y) = eqA x y
eq⟨+⟩ eqA eqB (Inr x) (Inr y) = eqB x y
eq⟨+⟩ eqA eqB x y = False
eq⟨×⟩ eqA eqB (x1, y1) (x2, y2) = eqA x1 x2 && eqB y1 y2
```

1.2 Coq

Coq is a system developed by Gérard Huet and Thierry Coquand (Bertot and Castéran, 2004) for verifying proofs. Coq is often called a *proof assistant* because it greatly eases the construction of a proof with the use of libraries and automated proof searching. There are numerous advantages to using Coq and other proof assistants like Isabelle (Paulson, 1989), Lego (Luo and Pollack, 1992) and PVS (Owre et al., 1996):

- The final proof will be machine verified. A proof done by hand is never guaranteed to be correct: some cases can be overlooked or there might be some hidden assumptions that we are not aware of. If the proof is done in Coq this is not possible, as Coq will ensure that all the cases are covered and any assumptions that need to be made have to be stated explicitly so that we are always aware of them.

- Coq will keep track of all the necessary subgoals that need to be proven. This is especially useful for long and complicated proofs where there might be a large number of subgoals.

- Proof libraries can be developed in Coq to assist in the construction of (parts of) proofs.

- In addition to proof libraries, Coq also supports specialized tactics. Tactics are commands that can be applied to a goal. Their effect is to produce a new, possibly empty, list of goals. If g is the input goal and $g_1 \dots g_k$ are the output goals, the tactic has an associated function that makes it possible to construct a solution of g from the solutions of goals g_i (Bertot and Castéran, 2004). Proofs about generic programs will have a common ground and we can use tactics to automate these parts of the proofs.

The formal system underlying the Coq proof assistant is the *Calculus of Constructions* (Coquand and Huet, 1988). The Calculus of Constructions is a dependently typed λ-calculus. Programs in the Calculus of Constructions correspond, via the Curry-Howard isomorphism, to proofs in higher-order impredicative logic. It is a close cousin of Martin-Löf type theory (which is, however, predicative) and is one of the most powerful constructive logics in use today.

An advantage of this approach is that proofs written in Coq are terms in a small and well understood mathematical calculus and can therefore be verified independently. Hence, it is possible to write a proof verifier: a relatively simple tool which can be verified in its entirety. Once verified, proofs in Coq are indeed guaranteed to be correct. In contrast, proofs written using other proof assistants, such as PVS or Sparkle (de Mol et al., 2002), are simply large bodies of instructions to those theorem provers. Therefore the correctness of a proof written in PVS or Sparkle relies on the correctness of the entire proof assistant. In particular, any future additional feature to one of these proof assistants may make it possible to prove invalid properties. In Coq, new features (especially new tactics) do not extend the core language so that generated proofs can still be verified against the Calculus of Constructions.

Furthermore, Coq takes care to maintain a distinction between programs that are proofs and programs that are meant to be executed. This mechanism is used to provide *program extraction*. Programs can be written in Coq, proven correct in Coq and then extracted to a general purpose programming language such as Haskell. The extracted program can then be compiled by an optimizing compiler and run. This means that the gap between the program which is proven correct and the program that is executed—often quite large—is minimal.

1.3 Proofs and Generic Haskell

To be able to reason about Generic Haskell programs, we need a way of representing such programs in the proof assistant Coq. In other words, we need an embedding of Generic Haskell in Coq. There are some important semantic differences between these two languages that need to be taken into consideration when trying to embed one in the other.

A function in Coq can be either inductive or coinductive, but it must always be *total*. For inductive functions Coq ensures the functions is total through a syntactic termination check. A coinductive function is considered total when it productive, and Coq verifies productivity through a syntactic guardedness check.

Haskell, on the other hand, allows functions to be partial. For example, we could write a function that filters out all elements that satisfy a given property P from a list. If the input list is infinite this function might go on forever without producing any result. While there is no direct translation from such a partial Haskell function to Coq, it is possible to encode partial functions in Coq. There are a number of ways to convert partial functions to total functions in Coq. For example, we could restrict the input type to lists of finite length (inductive datatypes). Another possibility is to use the partiality monad as described in (Capretta, 2005). We will give some pointers to other methods in Section 5.5.

Generic Haskell lives in the same world as Haskell and polytypic functions written in Generic Haskell can be partial. Programmers using our development will be able to write polytypic functions directly in the Coq proof assistant, and reason about those functions using all the tools available in Coq as well as the additional infrastructure for polytypic proofs that we provide. To reason about a partial polytypic function the programmer must use one of the available methods to encode this partiality as a total function in Coq.

1.4 Thesis Overview

Our work can be divided into three important stages: adding support for polytypic programming to Coq (Chapter 2), providing the necessary infrastructure to enable polytypic proofs about these programs (Chapter 3) and a way to deal with recursive datatypes and proofs (Chapter 4).

Chapter 2 is based on (Verbruggen et al., 2008) and discusses polytypic functions and their types. From a user's perspective, these functions strongly resemble the corresponding functions in Generic Haskell. Throughout this thesis we will use the polytypic `map` function and its type `Map` as our working example, both of which are defined in Section 2.4.

Once we have a polytypic definition for `map`, we can specialize it to any type in our generic view. For example, the specialization of `map` to the type of integers is simply the identity on integers:

```
Eval compute in specTerm tint map.
 = fun (z : Z) => z
 : specType tint Map
```

"`Eval compute in`" instructs Coq to compute the normal form of a term; `tint` is the "code" that corresponds to the type of integers, and `specTerm` and `specType` are our definitions of term and type specialization. In Section 2.6 we discuss term specialization. Coq reports the type of the result as the specialization of `Map` (the type of `map`) to `tint`; this evaluates to

```
Eval compute in specType tint Map.
 = Z → Z : Set
```

as expected (`Set` is Coq's name for kind $\star$). Type specialization is discussed in Section 2.5.

As a second more interesting example consider the type `fork`, which corresponds to the type $\Lambda A\ .\ A \times A$ (in Section 2.3 we discuss the encoding of `fork` in our generic view). To map a function across a term of this type we need to be given a function to map across its elements. Thus the specialization of `map` to `fork` is

```
Eval compute in specTerm fork map.
= fun (A B : Set) (f : A → B) (x : A × A) =>
   let (a, b) := x in (f a, f b)
: specType fork Map
```

The second stage of our development is discussed in Chapter 3, which is based on (Verbruggen et al., 2009). Polytypic properties are the Curry-Howard image of polytypic types, and are therefore defined by induction on kinds. As our working example we will use the property that `map` preserves identities:

$$map\ id = id$$

We define this property informally in Section 3.2.1 and give its formal definition in Section 3.2.2.

Just like polytypic types, we can specialize a property to a particular type in our universe, which is described in Section 3.4. For example, we can specialize the property that `map` preserves identities to the type `fork` to get the following property:

```
= ∀ (A : Set) (f : A → A) (Hx : ∀ x : A, f x = x)
     (p : A × A), map f p = p
```

We discuss the construction of a polytypic proof for such a property in Section 3.2.3. The interesting thing about polytypic proofs is that, just as for polytypic functions, the user only needs to provide the proofs for each of the type constants; we can then specialize this proof to any other type in our generic view. We discuss proof specialization and its difficulties in Section 3.5.

The last hurdle in our development is to introduce recursion, which is far from a trivial addition to the existing infrastructure. To avoid muddying the waters early on we will not discuss recursion in Chapters 2 and 3, but consider it as a separate problem in Chapter 4.

The approach to adding recursive datatypes and recursive proofs to our development closely follows the approach taken in Generic Haskell. Unfortunately, this will require some manual work on behalf of the programmer or the prover, and the approach is limited because of restrictions posed by the Coq guardedness checker. Nevertheless, our proof of concept demonstrates that the Generic Haskell approach is feasible in a formal setting. Making the approach more generic, or lifting the restrictions of the guardedness checker, is discussed in future work (Section 6.4).

We then finish off by discussing relevant related work in Chapter 5, followed by an overview of future work and conclusions in Chapter 6.

Throughout this thesis we will often use some form of syntactic sugar to make definitions more readable and easier to explain. The full definitions can always be found in the Coq sources (Verbruggen, 2009).

Chapter 2

Polytypic Programming

2.1 Introduction

The aim of our work is the development of an infrastructure in the proof assistant Coq for doing proofs over polytypic programs. In this chapter we present the first step towards this goal: the formalization of polytypic programming in Coq.

The approach to polytypic programming used in Generic Haskell was first introduced by Ralf Hinze (Hinze, 2000b,a) and has been implemented as a preprocessor for Haskell and as a language extension in Clean. It has since been recognized that in the context of a dependently typed language polytypic programming can be expressed entirely *within* the language and can be implemented simply as a library (for example, see Altenkirch and McBride, 2003). Our implementation too takes the form of a Coq library.

The core idea is that if f is a polytypic function of type F we can *specialize* f to an ordinary function $f\langle T\rangle$ over a datatype T. The type of $f\langle T\rangle$ is the specialization $F\langle T\rangle$ to the kind of T. Term specialization ($f\langle T\rangle$) is defined by induction on the structure of T; type specialization ($F\langle T\rangle$) is defined by induction on the *kind* of T.

The aim of this chapter is to provide an implementation of polytypic programming in Coq which is easily recognizable to programmers familiar with Generic Haskell or Generic Clean. In particular, our contributions are:

- We provide an infrastructure for defining polytypic functions and their types which is very similar to the infrastructure provided by Generic Haskell or Generic Clean (Section 2.4).
- We formalize term specialization and type specialization in Coq as defined in Generic Haskell/Clean. In particular, the definition of our type universe is identical (modulo syntactic differences).
- The definition in Coq has one very important benefit over the existing implementation in Generic Haskell. Since we use dependent types to specify that the result of `specTerm` T f must be of the form `specType` T F, our implementation is a formal *proof* that the term specialization $f\langle T\rangle$ must have type $F\langle T\rangle$.

The final point is important since it paves the way towards our ultimate goal of providing an infrastructure in Coq to prove properties about Generic Haskell style programs, which we discuss in Chapter 3. For this goal to succeed, we must have a definition of polytypic programming which is both fully formal and as close as possible to the definition in Generic Haskell or Generic Clean. This chapter provides such as definition. After a brief introduction to Coq in Section 2.2, we define our generic view (universe) in Section 2.3. We then show that the interface we provide to programmers in very similar to the Generic Haskell interface (Section 2.4) and give the formalization of type and term specialization in Sections 2.5 and 2.6.

We assume that the reader is familiar with Haskell, and has at least some cursory knowledge of Generic Haskell or Generic Clean. We will not assume any prior knowledge of Coq.

This chapter is an expanded version of (Verbruggen et al., 2008), and the Coq sources associated with the formalization described in this chapter can be found online (Verbruggen, 2009).

2.2 Coq

Before we delve into our formalization of polytypic programming, we will first give a brief overview of Coq. Coq is a proof assistant developed in Inria (Bertot and Castéran, 2004) based on the calculus of constructions (higher-order predicate logic) extended with inductive and coinductive datatypes and an infinite hierarchy of universes. The examples we discuss in this section are all part of our development, and will be referred to in the rest of the chapter.

2.2.1 Dependent Types

The calculus of constructions (Coquand and Huet, 1988) is a dependently typed lambda calculus. This means that types are first-class and can be passed as arguments to functions or returned as results. For example, we can define a function `tupleT` A n which constructs the type of homogeneous tuples containing n elements of type A[1]

$$\underbrace{A \times A \times \cdots \times A}_{n}$$

as

```
Fixpoint tupleT (A : Type) (n : nat) {struct n} : Type :=
 match n with
 | O => unit
 | S m => A * tupleT A m
 end.
```

[1]For each definition associated with `tupleT` we have an equivalent definition associated with `tupleS`, where the only difference is that the type A of the elements has kind `Set` instead of kind `Type`.

`Fixpoint` introduces a recursive definition, and `match` denotes pattern matching (comparable to `case` in Haskell). Since all functions in Coq must terminate, recursive functions must be defined by structural induction on one of its arguments, denoted by the keyword `struct`. The termination checker in Coq will verify that all recursive calls are made to structurally smaller arguments. Notice that for the case of $n = 0$ we return the unit type (`()` in Haskell) which has one element denoted by `tt`. Thus we will use `tt` to denote the empty tuple.

To define a function that returns the ith element from an n-tuple, we must first define a datatype that describes the set of valid indices $\{0, 1, \ldots, n-1\}$. This datatype is known as `index` and is defined as

```
Fixpoint index (n : nat) : Set :=
 match n with
 | O => Empty_set
 | S m => option (index m)
 end.
```

The `option` type can be compared to the `Maybe` monad in Haskell:

```
Inductive option (A : Type) : Type :=
 | Some : A → option A
 | None : option A.
```

`Inductive` introduces an inductive datatype. The syntax can be compared to the syntax for GADTs in Haskell (Peyton Jones et al., 2006): we specify the kind of the datatype and list the types of the constructors. The `option` type has two constructors: `None` and `Some`.

The `index` type is parametrized by a natural number—a term, not a type—and is therefore a so-called *dependent* datatype: a type depending on a term. For anyone not familiar with Coq and dependently typed programming languages, the `index` type might seem a little strange. One way to view it is as a family of types:

For $n = 0$ we get the empty type `Empty_set`, so `index 0` represents the type containing no elements; i.e. for a tuple of length 0 there are no possible indices.

For $n = 1$ we get the type `option (index 0)`, which further unrolls to `option Empty_set`. There are two ways to construct an element of type `option` A. To use the `Some` constructor we need to provide an element of type A, since `Empty_set` is the empty type no such element exists. That leaves us with the second constructor which can always be applied to give us the element `None : option Empty_set`. In other words, for a tuple of length 1 there is one possible index which we call `None`.

For $n = 2$ we can unroll to get the type `option (option Empty_set)`. We can again apply the second constructor to get the element `None : option (option Empty_set)`, but we can now also apply the first constructor since we can supply it with an element of the correct type. In this case we have $A =$ `option Empty_set`, and we have seen already that this contains the single element `None`, so we can construct the element `Some None : option (option Empty_set`. This gives us two possible indices for tuples of length 2.

This pattern continues for all values of n: we take the $n-1$ elements of type `index` $(n-1)$, apply the `Some` constructor to all of them to get $n-1$ elements of type `index` n and add in `None : index` n to get a total of n elements of type `index` n. Throughout this thesis we will often replace the syntax of indices by natural numbers for readability, where $0 =$ `None`, $1 =$ `Some None`, etc.

Given the datatype `index` we can write the indexing operator that takes the ith element from an n-tuple as

```
Fixpoint getT (A : Type) (n : nat) {struct n}
  : index n → tupleT A n → A :=
 match n return index n → tupleT A n → A with
 | O => fun i _ => match i with end
 | S m => fun i tup => match i with
                      | None => fst tup
                      | Some i' => getT A m i' (snd tup)
                      end
 end.
```

The syntax "`match` e `in` T `return` σ `with`" denotes a dependent pattern match on $e : T$, where the type of each branch has type σ (which may depend on both e and T, but in this case only depends on e so we can leave the `in` clause empty). The use of an underscore in the case for $n = 0$ indicates an implicit variable that can be automatically inferred by Coq.

The case for $n = 0$ is slightly obscure. Since $n = 0$, we know that i must have type `index 0`, which we have already said is the type containing no elements. Matching on i will therefore not generate any possible cases and we end up with an empty `match` construct. This prevents us trying to extract an element from an empty tuple. Also, since the index i must always be within bounds due to its type, `getT` is a total function.

2.2.2 Proofs

From a logical perspective, Coq's language corresponds to constructive higher-order predicate logic where every program in Coq denotes the proof of its type. This fascinating result is known as the Curry-Howard isomorphism, but a discussion of this topic would take us too far afield; we refer the reader to the excellent textbook by Sørensen and Urzyczyn (2006) instead.

For simple cases we can write these proofs as programs. For example, here is a proof of modus ponens:

```
Lemma MP : ∀ (A B : Prop), A → (A → B) → B.
Proof (fun (A B : Prop) (a : A) (f : A → B) => f a).
```

"`Lemma` ... `Proof`" is alternative syntax for "`Definition` ... :=" to make it clear that we are writing a proof rather than a program. Of course, there really is no distinction and this is syntactic sugar only. We could also write

```
Definition MP : ∀ (A B : Prop), A → (A → B) → B :=
 fun (A B : Prop) (a : A) (f : A → B) => f a.
```

The two definitions of `MP` are indistinguishable. Coq does, however, make a formal distinction between terms that are "programs" and terms that are "proofs" in the type system. The type of a program (say, `nat`) lives in `Set`, as we saw above. The type of a proof (that is, a proposition such as $1 = 1$) lives in `Prop` (both `Set` and `Prop` live in $\texttt{Type}_0$). The reason for the two different sorts is that Coq supports *program extraction*: Coq can extract all the computational content (that is, keep the programs but strip the proofs) to be exported to OCaml or Haskell for efficient compilation.

For more complicated proofs, however, writing proofs by hand (as "programs") becomes difficult. Instead, we can make use of *tactics*. Tactics are small programs that can search for proofs in a particular domain. The use of tactics enables *proof automation*, where Coq can handle most of the more mundane parts of our proofs automatically. This is a huge help in any realistic proof. One of the simplest tactics is `auto`, which attempts to solve the proof by repeated application of the currently available hypotheses. Other tactics include tactics for induction (i.e., recursion), inversion, arithmetic, etc. Moreover, Coq supports a language called *Ltac* in which custom tactics can be written. All tactics will search for proofs, and then *return* a proof if one can be found—which will be verified by Coq. This means that a "rogue" tactic cannot compromise the soundness of the system.

We will not make any significant use of tactics in our development, so we refer the reader to (Bertot and Castéran, 2004) for more information. However, the support for tactics and proof automation is an important reason for choosing Coq for our work (reasoning about polytypic programs) since they will ease the burden on users of our system; see also Section 6.1.1.

2.2.3 Universe Inconsistencies

In the next few sections we will give a short overview of some of the major stumbling blocks in dealing with Coq. In this section we will detail how we avoided universe inconsistencies in our development.

One definition that we will need later in our proof is a characterization of heterogeneous tuples, in which every element has a different type. One natural way we might consider is to define a function which given a tuple of types

$$(A, B, C)$$

constructs the type

$$A \times B \times C$$

We can define this function as

```
Fixpoint gtupleT (n : nat) : tupleT Type n → Type :=
 match n return tupleT Type n → Type with
 | 0   => fun _ => unit
 | S m => fun tup => fst tup * gtupleT m (snd tup)
 end.
```

This definition works fine in most cases. However, if we want to construct a heterogeneous tuple where the elements themselves are tuples, i.e a heterogeneous tuple of the form

$$\texttt{tupleT}\ A_1\ m_1 \times \texttt{tupleT}\ A_2\ m_2 \times \cdots$$

Coq will come back with the uninformative error

```
Universe inconsistency
```

To understand this error, we must know a little more about universes in Coq. Like in Haskell, the natural number "5" has type `nat`. Like in Haskell, the type `nat` has kind (or type) `Set` (`Set` is called $\star$ in Haskell). Unlike in Haskell, however, this hierarchy continues ad infinitum: `Set` has type $\texttt{Type}_0$, $\texttt{Type}_0$ has type $\texttt{Type}_1$, and generally $\texttt{Type}_i$ has type $\texttt{Type}_{i+1}$. Moreover, there is a coercion rule that if $T : \texttt{Type}_i$ then $T : \texttt{Type}_j$ for any $j \geq i$. This *stratification* of `Type` prevents the encoding of logical paradoxes (e.g., Hurkens, 1995).

The user cannot assign these universe levels manually, which is why we simply wrote `Type` in the examples above. Coq attempts to assign a suitable level to each occurrence of `Type`, infers the constraints on these levels, and verifies that there are no inconsistencies. For `tupleT` we have

$$\texttt{tupleT} : \texttt{Type}_i \to \texttt{nat} \to \texttt{Type}_j \quad (i \leq j)$$

The constraint $(i \leq j)$ comes from the fact that the first argument, $A : \texttt{Type}_i$, is used to construct the new type $A \times A \times \cdots \times A : \texttt{Type}_j$.

Now consider what happens when we try to define our tuple of tuple types. The elements of the tuple are the result of `tupleT` and therefore have type $\texttt{Type}_j$. The constructed type itself must then have type

$$(\texttt{tupleT}\ A\ m : \texttt{Type}_j, \ldots) : \texttt{tupleT}\ \texttt{Type}_j\ n$$

Since we pass $\texttt{Type}_j$ as the first argument to `tupleT`, and we have said that this first argument has type $\texttt{Type}_i$, we must have $\texttt{Type}_j : \texttt{Type}_i$ which will hold only if $j < i$. But the constraints $i \leq j$ and $j < i$ cannot both be satisfied, and Coq reports a universe inconsistency: there is no suitable assignment that does not result in an inconsistency.

The problem is that Coq does not support universe polymorphism (Harper and Pollack, 1991). A work-around would be to duplicate the definition of `tupleT` which would then have type $\texttt{Type}_{i'} \rightarrow \texttt{nat} \rightarrow \texttt{Type}_{j'}$. This would change the constraints to $j' < i, i \leq j$ and $i' \leq j'$, thus solving the inconsistency. This is, however, not a very elegant solution, especially since it would lead to further code duplication elsewhere. Fortunately, we can follow Morris et al. (2007b) and give an alternative definition of heterogeneous tuples which avoids universe inconsistency without the need for duplication (Morris et al. refer to this operator as the modality $\Box$). Given a tuple

$$(x, y, z)$$

of elements of some type A and a function $f : A \rightarrow \texttt{Type}$, we construct the type

$$fx \times fy \times fz$$

This is implemented as

```
Fixpoint gtupleT (A : Set) (n : nat) (f : A → Type)
   : tupleS A n → Type :=
  match n return tupleS A n → Type with
  | O   => fun _ => unit
  | S m => fun tup => f (fst tup) * gtupleT m f (snd tup)
  end.
```

While this definition is not formally equivalent, it is equally suitable for our purposes and avoids the universe inconsistency. The indexing operator associated with `gtupleT` takes the following form:

```
ggetT : ∀ (A : Set) (n : nat) (f : A → Type) (i : index n)
  (tup : tupleS A n), gtupleT f tup → f (getS i tup)
```

2.2.4 Equality

In Coq, equality is an inductive datatype with exactly one constructor (`refl_equal`) which says that $x = x$. Therefore, the expression $x = y$ is true if and only if both x and y reduce to the same term, under the usual β, δ, ι and ζ-reductions (see Section 4.3.2 for an overview of these reductions).

Equality of Functions

This definition of equality implies the absence of an extensionality principle. Two functions to sort the elements of a list (e.g. quicksort and mergesort) will be regarded as different functions, even though they produce the same output for every possible input. This lack of extensionality is not unreasonable: after all, quicksort and mergesort are not the same function.

When we start defining properties about our polytypic functions in Chapter 3 this definition of equality becomes important. The property we would like to define states that the polytypic `map` function preserves composition, i.e.

$$\text{map } f \circ \text{map } g = \text{map } (f \circ g)$$

However, these two functions are constructed differently and because extensionality does not hold in Coq they can never be equal. Fortunately, while we cannot prove that the two functions are equal, we *can* prove that the result of the functions is the same given the same input:

$$(\text{map } f \circ \text{map } g)\ x = \text{map } (f \circ g)\ x$$

We will define all properties in Chapter 3 in this point-wise style to avoid the problem of extensionality.

Extensionality as an Axiom

Another solution we could have adopted is to add an axiom of extensionality to Coq, which does not lead to an inconsistent theory (The Coq Development Team, 2008a, Section 5.2):

```
Axiom extensionality : ∀ (A B : Set) (f g : A → B),
 (∀ x : A, f x = g x) → f = g.
```

We have opted not to add extensionality in this thesis because our development can be done entirely without assuming the axiom. However, if desired, the axiom can be added and polytypic properties and proofs can be modified accordingly.

Leibniz Equality

Another common notion of equality is *Leibniz Equality*, which can be defined in Coq as follows:

```
Definition Leibniz_eq (A : Set) (x y : A) :=
 ∀ (B : Set) (h : A → B), h x = h y.
```

It can be shown that when extensionality is added as an axiom in Coq, Leibniz equality coincides with pointwise equality (see Figure 2.1). Alternatively, it is possible to add an axiom stating that pointwise equality implies Leibniz equality, from which extensionality can be shown (see Figure 2.2). Therefore, the two axioms are equivalent.

Heterogeneous Equality

Another stumbling block regarding equality was that we often had to prove that two elements $a : A$ and $b : B$ were equal, where A and B are provably equal but not syntactically equal. This type of equality is usually referred to as *heterogeneous* or *John Major* equality. We will come back to this in Section 3.3, where we also explain the difficulties in doing proofs about heterogeneous equalities.

```
Axiom extensionality : ∀ (A B : Set) (f g : A → B),
 (∀ (x : A), f x = g x) → f = g.

Lemma Leibniz_fn : ∀ (A B : Set) (f g : A → B),
 (∀ (x : A), f x = g x) → Leibniz_eq f g.
Proof.
 intros.
 unfold Leibniz_eq ; intros.
 assert (f = g) by apply (extensionality _ _ H).
 rewrite H0.
 reflexivity.
Qed.

Remark Leibniz_eq_to_pointwise : ∀ (A B : Set) (f g : A → B),
 Leibniz_eq f g → (∀ (x : A), f x = g x).
Proof.
 intros.
 unfold Leibniz_eq in H.
 assert (f = g) by (change (id f = id g) ; apply H).
 rewrite H0.
 reflexivity.
Qed.
```

Figure 2.1: Extensionality to Leibniz Equality

```
Axiom Leibniz_fn : ∀ (A B : Set) (f g : A → B),
 (∀ (x : A), f x = g x) → Leibniz_eq f g.

Lemma extensionality : ∀ (A B : Set) (f g : A → B),
 (∀ (x : A), f x = g x) → f = g.
Proof.
 intros.
 assert (Leibniz_eq f g) by apply (Leibniz_fn _ _ H).
 unfold Leibniz_eq in H0.
 change (id f = id g) ; apply H0.
Qed.

Remark extensional_to_pointwise : ∀ (A B : Set) (f g : A → B),
 f = g → (∀ (x : A), f x = g x).
Proof.
 intros.
 rewrite H.
 reflexivity.
Qed.
```

Figure 2.2: Leibniz Equality to Extensionality

2.2.5 Coinduction

Where all recursive functions and datatypes must be shown to be terminating, all corecursive functions and datatypes need to productive. Productivity guarantees that in each step the function takes at least one new constructor will be produced. Even if we can never produce the complete result (which could be an infinite list), at least we know that each step will give us a new piece of the puzzle.

Coq guarantees productivity of corecursive functions and datatypes using a syntactic guardedness check. In practise this means that every corecursive call must be guarded by a constructor of the output type of the function. For example, we can represent the conatural numbers as

```
CoInductive coN : Set :=
 | coZ : coN
 | coS : coN → coN.
```

and define infinity as

```
CoFixpoint inf := coS inf.
```

where the corecursive call to `inf` is clearly guarded by the constructor `coS` of the datatype `coN`.

The second condition is that the constructor guarding the corecursive call cannot in turn be passed as an argument to another function if that function cannot be unrolled by Coq. In Figure 2.3 we use a simple example to show that a guarded call passed as an argument to an unknown function can become unguarded. A corecursive function is regarded as an unknown function, so any argument passed to it will be considered unguarded.

This syntactic guardedness check is quite restrictive, especially considering the fact that Coq cannot unroll any coinductive functions and must therefore treat them as unknown functions during the guardedness check. We need to be careful in constructing our proofs to avoid passing the results of corecursive calls as arguments to coinductive (or otherwise "unknown") functions.

In Chapter 4 we will show how we can use corecursive functions to deal with the specialization of polytypic functions (and proofs) to (co)recursive datatypes. In Section 4.5 we show that a small change in the construction of a particular proof can make that proof unguarded.

2.3 Definition of the Generic View

A generic view is a set of *codes* that represent the datatypes that can be used as a target for specialization of polytypic functions. For example, if we want to specialize the polytypic map function to the product type (`prod` in Coq), we must pass the code for `prod` (`tprod`), as well as the definition of `map` itself, as arguments to term specialization.

```
CoInductive C : Set :=
 | MkC : C → C.

(* The corecursive call is guarded by the MkC constructor *)
CoFixpoint guarded_fn1 : C := MkC guarded_fn1.

(* This does not change the guardedness of its argument *)
Definition leave_guarded (c : C) : C :=
 match c with
 | MkC c' => MkC c'
 end.

(* The corecursive call is again guarded by MkC, but is then
   passed as an argument to leave_guarded. Since we can unroll
   the definition of leave_guarded, we can see that this does
   not affect guardedness. *)
CoFixpoint guarded_fn2 : C := leave_guarded (MkC guarded_fn2).

(* Stripping the top-level constructor *)
Definition unguard (c : C) : C :=
 match c with
 | MkC c' => c'
 end.

(* Even though the corecursive call is guarded by MkC, this
   constructor is stripped by the function unguard, leaving
   it unguarded. This function will not be accepted by Coq.
CoFixpoint not_guarded1 : C := unguard (MkC not_guarded1).
*)

(* An undefined function. *)
Variable unknown_fn : C → C.

(* Since Coq cannot unroll an unknown function it cannot accept
   this definition - the unknown function could be unguard.
   Therefore it is assumed that all guarded corecursive calls
   become unguarded when passed to an unknown function.
CoFixpoint not_guarded2 : C := unknown_fn (MkC not_guarded2).
*)

(* Coinductive functions cannot be unrolled in Coq and are thus
   treated as unknown function by the guardedness checker. *)
```

Figure 2.3: Guarded Recursive Call Becomes Unguarded

However, the result of term specialization should be a function on the product datatype proper (`prod`). This means that we must define a mapping from codes in the generic view to ordinary Coq datatypes. Such a mapping is known as a *decoder*. Similarly we need to encode kinds and define the associated decoder. The definitions for our generic view and its decoders are listed in Figure 2.4.

2.3.1 Kinding Derivations

In our definition of the generic view we do not define a datatype that encodes the *grammar* of types, but rather encode kinding derivations to make sure that only well-kinded types can be represented. An element

$$T : \texttt{type}\ nv\ ek\ k$$

is a type of kind k with at most nv free variables, whose kinds are defined in the kind environment ek. This corresponds to a kinding derivation

$$ek \vdash T : k$$

The type of the environment ek is `envk` nv, which is an nv-tuple of kinds.

As an example, the rule for lambda abstraction encodes the kinding derivation

$$\frac{(k_1, ek) \vdash T : k_2}{ek \vdash \Lambda T : k_1 \to k_2}\ \textsc{Lam}$$

Note that we are using De Bruijn indices to represent variables (de Bruijn, 1972). The indices in a type of nv free variables are of type `index` nv (see Section 2.2.1), which guarantees that no indices can be out of bounds.

2.3.2 Decoding Kinds

The decoder for kinds is straight-forward, but there is a subtlety with the choice of `Set` as the decoding of kind $\star$. In the specialization of the arrow kind (Section 2.5), we will construct types of the form

$$(\forall(\alpha : \texttt{decK star}).\ \ldots) : \texttt{decK star}$$

Since the bound variable (α) ranges over the very type that is defined, the type of α must be impredicative. As we have seen in Section 2.2.3, `Type` in Coq is not impredicative (but stratified) and returning `Type` for the decoding of kind $\star$ will result in a universe inconsistency when we subsequently attempt to define type specialization. Hence we choose `Set` instead, enabling the impredicative `Set` option[2].

[2]`Set` was impredicative in Coq by default before version 8; this was changed mainly to support classical reasoning. We will not use classical reasoning, however, and so making `Set` impredicative does not compromise the soundness of our proofs (see The Coq Development Team, 2008a,b).

Another way to solve this problem is to stratify kinds themselves, i.e. to assign different levels to kind $\star$ depending on nesting depth. We would then get something of the form

$$(\forall(\alpha : \texttt{decK star}_i)\ \texttt{.}\ \ldots) : \texttt{decK star}_j$$

Here we can use `Type` as the decoding of kind $\star$, with different universe levels assigned to the different nesting levels of the kinds. We have opted to use impredicative `Set` instead because we did not want to complicate the kind universe. It would be interesting to see how the infrastructure would change if we use stratified kinds.

2.3.3 Decoding Types

The decoder for types is more involved. To decode a type T with *nv* free variables, we must know the decoded types of the free variables in T. Hence, we need an environment *et* of type `envt` that associates a decoded type T_i with every free variable i in T. Since the type of T_i (its kind, if you prefer) depends on the type of i, each element in *et* has a different type. We therefore calculate `envt` from the kind environment *ek*:

```
Definition envt nv (ek:envk nv) := gtupleT decK ek.
```

using the heterogeneous tuple described in Section 2.2.3.

We have already introduced two different environments (*ek* : `envk` *nv* and *et* : `envt` *nv ek*) and we will need two more in the remainder of this chapter. As it may be difficult to keep track of so many different environments we provide an overview of the definitions and their purpose in Figure 2.5.

Armed with the type environment *et* we can define the decoder for types as shown in Figure 2.4. Type constants map to their Coq counterparts, variables map to the corresponding elements in the environment *et*, application maps to Coq type application and lambda abstraction maps to Coq type-level functions. To decode the body of a lambda abstraction we must add the type of the formal parameter to the type environment.

2.3.4 Example Types

In this section we will consider some examples of types, defined as codes in our generic view with the associated decoding. Consider the type `fork`: $\Lambda A\ .\ A \times A$, which we can encode in our generic view as

```
Definition fork : closed_type (karr star star) :=
 let var := tvar 1 (star, tt) in
  tlam (var None × var None).
```

The type of `fork` tells us that it is a closed type of kind $\star \to \star$. Unfortunately, the notation (`var` i represents the ith variable) is slightly heavy, but the addition of syntactic sugar is future work. We can decode `fork` to an actual Coq type using our type decoder:

```
Eval compute in decT fork tt.
```

```
(* Codes for kinds *)
Inductive kind : Set :=
 | star : kind
 | karr : kind → kind → kind.

(* Decoder for kinds *)
Fixpoint decK (k : kind) : Type :=
 match k with
 | star => Set
 | karr k1 k2 => decK k1 → decK k2
 end.

(* Grammar for type constants *)
Inductive type_constant : kind → Set :=
 | tc_unit : type_constant star
 | tc_int  : type_constant star
 | tc_prod : type_constant (karr star (karr star star))
 | tc_sum  : type_constant (karr star (karr star star)).

(* Codes for types *)
Inductive type : ∀ (nv : nat), envk nv → kind → Set :=
 | tconst : ∀ nv ek k, type_constant k → type nv ek k
 | tvar   : ∀ nv ek i, type nv ek (getS i ek)
 | tapp   : ∀ nv ek k1 k2,
     type nv ek (karr k1 k2) → type nv ek k1 → type nv ek k2
 | tlam   : ∀ nv ek k1 k2,
     type (S nv) (k1, ek) k2 → type nv ek (karr k1 k2).

(* Syntactic sugar for types with no free variables *)
Definition closed_type (k : kind) : Set := type 0 tt k.

(* Syntactic sugar for type constants *)
Definition tunit := tconst 0 tt tc_unit.
Definition tint  := tconst 0 tt tc_int.
Definition tprod := tconst 0 tt tc_prod.
Definition tsum  := tconst 0 tt tc_sum

(* Decoder for types *)
Fixpoint decT (nv : nat) (k : kind) (ek : envk nv) (t : type nv ek k)
  {struct t} : envt nv ek → decK k :=
 match t in type nv ek k return envt nv ek → decK k with
 | tconst nv ek k tc      =>
    fun et => match tc in type_constant k return decK k with
              | tc_unit => unit
              | tc_int => Z
              | tc_prod => prod_set
              | tc_sum => sum_set
              end
 | tvar nv ek i           => fun et => ggetT i et
 | tapp nv ek k1 k2 t1 t2 => fun et => (decT t1 et) (decT t2 et)
 | tlam nv ek k1 k2 t'    => fun et arg => (decT t' (arg, et))
end.
```

Figure 2.4: Generic View and Decoders

```
(* Kind environment:
   associates a kind with each free variable *)
Definition envk (nv : nat) : Set := tupleS kind nv.

(* Type environment:
   associates a Coq type with each free variable *)
Definition envt (nv : nat) (ek : envk nv) := gtupleT decK ek.

(* Environment of the form ((a1,b1,...),(a2,b2,...),...,(anp,bnp,...))
   to keep track of free variable replacements
   in type and term specialization *)
Definition envts (np nv : nat) (ek : envk nv) :=
  tupleT (envt nv ek) np.

(* Environment containing the functions associated with
   free variables in term specialization *)
Definition envf (np nv : nat) (ek : envk nv)
  (Pt : PolyType np) (ets : envts np nv ek) :=
 gtupleS (fun i => specType' (tvar nv ek i) Pt ets)
         (elements_of_index nv).
```

Figure 2.5: Overview of Environments

```
 = fun A : Set => A × A
 : decK (karr star star)
```

and similarly we can decode its kind as

```
Eval compute in decK (karr star star).
 = Set → Set : Type
```

As a slightly more complicated example let us consider the type maybe_prod, which has kind $\star \to \star \to \star$, and represents the type that is either the unit type or a product: $\Lambda A \,.\, \Lambda B \,.\, 1 + A \times B$, defined in Coq as

```
Definition maybe_prod
  : closed_type (karr star (karr star star)) :=
 let var := tvar 2 (star, (star, tt)) in
  tlam (tlam (1 + (var (Some None) × var None))).
```

We can decode this type to

```
Eval compute in decT maybe_prod tt.
 = fun A B : Set => unit + A × B
```

which gives us a Coq function in two arguments of type Set. Note that here unit refers to the predefined unit type in Coq, and $\times$ and $+$ refer to the predefined product and sum types.

Finally, we will show the code for apply : $(\star \to \star) \to \star \to \star$, which takes a type constructor $F : \star \to \star$ and a type $A : \star$ and applies F to A:

```
Definition apply :
 closed_type (karr (karr star star) (karr star star)) :=
let var := tvar 2 (star, (karr star star, tt)) in
 tlam (tlam (var (Some None) @ var  None)).
```

The decoding will again be an actual Coq function, using proper Coq application to apply F to A:

```
Eval compute in decT apply tt.
 = fun (F : Set → Set) (A : Set) => F A
```

2.4 Defining Polytypic Functions

In this section we will explain how polytypic functions and their types can be defined using our library. We think that readers familiar with Generic Haskell or Generic Clean will experience a comforting familiarity reading our definitions; we will explain specifics pertaining to Coq as they arise.

2.4.1 Polytypic Types

The type of a polytypic function is a (type-level) function which, given *np* arguments, constructs a type of kind $\star$:

```
Record PolyType (np : nat) : Type := polyType {
 typeKindStar : nary_fn np (decK star) (decK star)
}.
```

The `Record` keyword introduces a record of named fields; the difference between records in Coq and records in Haskell is that records in Coq can be dependent. `PolyType` has one parameter (`np`) and one field (`typeKindStar`) of type `nary_fn np Set Set`. A term `nary_fn` $n\ A\ B$ denotes the type

$$\underbrace{A \to \ldots \to A}_{n} \to B$$

We will refer to `np` as the number of arguments of the polytypic function (it does *not* refer to the number of arguments of the specialized function, which varies with the kind of the target type).

As readers who are familiar with polytypic programming will know, `map` is a polytypic function of two arguments; its type `Map` is

```
Definition Map : PolyType 2 :=
 polyType 2 (fun A B => A → B).
```

The type of the polytypic function describes the type of the operation that gets performed by the polytypic function at the elements; in this case, `map` transforms elements of type A to elements of type B. Specialization of a polytypic function uniformly lifts the operation on elements to an operation on structures containing elements. The specialization of the *type* of the polytypic function describes the type of the lifted operation.

2.4.2 Polytypic Functions

To define a polytypic function, the user only needs to provide the definition for the type constants; term specialization then takes care of the remaining types. A nice feature of an implementation of polytypic programming in a dependently typed language is that the definition of a polytypic function is simply another record, making it very clear to the user what information is required to define a polytypic function. Reifying a polytypic function as a record also makes polytypic functions first-class, i.e. they become ordinary objects in the host language and can be passed as arguments and returned as results of other functions. The record representing polytypic functions is defined as follows:

```
Record PolyFn (np : nat) (Pt : PolyType np) : Type :=
 polyFn {
  punit : specType tunit Pt ;
  pint  : specType tint Pt ;
  pprod : specType tprod Pt ;
  psum  : specType tsum Pt
}.
```

In words, a polytypic function has a (polytypic) type of `np` arguments, and contains definitions for the type constants `tunit`, `tint`, `tprod` and `tsum` (for simplicity's sake, we do not consider other type constants). The type of these fields is determined by type specialization (explained in Section 2.5), which ensures that users cannot define ill-typed polytypic functions. For the specific case of `Map`, this simplifies to

```
punit : unit → unit
pint  : Z → Z
pprod : ∀ (A B : Set), (A → B) →
        ∀ (C D : Set), (C → D) → A × C → B × D
psum  : ∀ (A B : Set), (A → B) →
        ∀ (C D : Set), (C → D) → A + C → B + D
```

We can ask Coq to simplify these types for us, which is a great help when defining the polytypic function. We can now define polytypic `map` as

```
Definition map : PolyFn Map :=
 polyFn Map
  (fun (u : unit) => u)
  (fun (z : Z) => z)
  (fun (A B : Set) (f : A → B)
       (C D : Set) (g : C → D) (x : A × C) =>
    let (a, c) := x in (f a, g c))
  (fun (A B : Set) (f : A → B)
       (C D : Set) (g : C → D) (x : A + C) =>
    match x with
    | inl a => inl _ (f a)
    | inr c => inr _ (g c)
    end).
```

```
(* Specialize polytypic type Pt to kind k *)
Fixpoint kit (k : kind) (np : nat) (Pt : PolyType np) {struct k}
  : tupleT (decK k) np → decK star :=
 match k return tupleT (decK k) np → decK star with
 | star       => uncurry (typeKindStar Pt)
 | karr k1 k2 => fun tup => quantify_tuple
    (fun As => kit k1 Pt As → kit k2 Pt (apply_tupleT tup As))
 end.

(* Apply kit k Pt to the tuple \flrtup{t}{\np} *)
Definition specType' (np nv : nat) (k : kind) (ek : envk nv)
  (t : type nv ek k) (Pt : PolyType np) (ets : envts np nv ek)
  : decK star :=
 kit k Pt (replace_fvs t ets).

(* Type specialization for closed types *)
Definition specType (np : nat) (k : kind) (t : closed_type k)
  (Pt : PolyType np) : decK star :=
 specType' t Pt (ets_tt np).
```

Figure 2.6: Type Specialization

This is virtually identical to the definition we would give in Generic Haskell or Generic Clean (see Section 5.3.2 for the Generic Haskell definition), with one exception perhaps: since Coq is explicitly typed, the fields of the polytypic function take explicit type arguments. For example, in Haskell we would write the case for the product type as

```
λf -> λg -> λx -> let (a, c) = x in (f a, g c)
```

2.5 Type Specialization

As we saw in Section 2.4, a polytypic function `map` has a polytypic type `Map`, and the specialization `specTerm` T `map` of `map` to a type T has the specialized type `specType` T `Map`. In this section we explain how to define

```
specType : ∀ np k, closed_type k → PolyType np → Set
```

The full definition of type specialization is given in Figure 2.6.

Type specialization is a two-phase process. We first define the kind-indexed type `kit k Map` by induction on `k`. Informally, this can be defined as

$$\mathtt{Pt}\langle k\rangle : k \to \cdots \to k \to \star$$

$$\mathtt{Pt}\langle \star\rangle\ T_1 \ldots T_{np} = \textit{(user defined)}$$

$$\mathtt{Pt}\langle k_1 \to k_2\rangle\ T_1 \ldots T_{np} = \forall A_1 \ldots A_{np}\ .$$

$$\quad \mathtt{Pt}\langle k_1\rangle\ A_1 \ldots A_{np} \to \mathtt{Pt}\langle k_2\rangle\ (T_1\ A_1) \ldots (T_{np}\ A_{np})$$

The case for kind $\star$ is supplied by the user (`PolyType`, see Section 2.4). We can rewrite the case for arrow kinds as

$$\texttt{Pt}\langle k_1 \to k_2 \rangle = \Lambda T_1 \ldots T_{np} \,.\, \forall A_1 \ldots A_{np} \,.\, (\ldots)$$

to make it more obvious that we must return a type-level function which, given np arguments, returns a universally quantified type. It is however difficult to give a recursive definition of this type; a seemingly trivial but very helpful insight is that it is much easier to work with an uncurried form (it is interesting to note that Altenkirch and McBride (2003) come to the same conclusion). This gives us the following definition of $\texttt{Pt}\langle k_1 \to k_2 \rangle$:[3]

$$\begin{aligned}&\Lambda(T_1, \ldots, T_{np}) \,.\, \forall A_1 \ldots A_{np} \,.\\ &\qquad\qquad \texttt{Pt}\langle k_1 \rangle\ (A_1, \ldots, A_{np}) \to \texttt{Pt}\langle k_2 \rangle\ (T_1\ A_1, \ldots, T_{np}\ A_{np})\end{aligned}$$

To construct this function we first construct the function where both the T's and A's are uncurried:

$$\begin{aligned}&\Lambda(T_1, \ldots, T_{np}) \,.\, \Lambda(A_1, \ldots, A_{np}) \,.\\ &\qquad\qquad \texttt{Pt}\langle k_1 \rangle\ (A_1, \ldots, A_{np}) \to \texttt{Pt}\langle k_2 \rangle\ (T_1\ A_1, \ldots, T_{np}\ A_{np})\end{aligned}$$

This can be translated to the correct type using the function `quantify_tuple`, which takes a function of the form

$$\Lambda(A_1, \ldots, A_{np}) \,.\, T$$

to the universally quantified type

$$\forall A_1 \ldots A_{np} \,.\, T$$

This function can be implemented as follows:

```
Fixpoint quantify_tuple (A : Type) (n : nat)
  : (tupleT A n → Set) → Set :=
 match n return (tupleT A n → Set) → Set with
 | O => fun f => f tt
 | S m => fun f =>
    ∀ a : A, quantify_tuple A m (fun As => f (a, As))
 end.
```

Paraphrasing, `kit k Pt` constructs a type that calculates the required specialized type given a tuple $(T_1, \ldots, T_{np})$; the second step in type specialization is therefore to construct this tuple. Hinze states that specialization of a polytypic function `pfn` of type `Pt` to a type T has type

[3]It is possible to uncurry the first part of the definition because the function is never partially applied. We could also leave the second set of arguments (the A's) uncurried, but this generates unreadable types.

$$\texttt{pfn}\langle T : k\rangle : \texttt{Pt}\langle k\rangle\ (\lfloor T\rfloor_1, \ldots, \lfloor T\rfloor_{np})$$

The definition of the floor operator $\lfloor\,\rfloor_i$ is slightly involved, so we will consider an example first[4]. The type `Map` specialized to the datatype $T_{\text{ex}} = \Lambda A\ B\ C\ .\ A + B \times C$ should be

$$(A_1 \to A_2) \to (B_1 \to B_2) \to (C_1 \to C_2) \to T_{\text{ex}}\ A_1\ B_1\ C_1 \to T_{\text{ex}}\ A_2\ B_2\ C_2$$

Recall that the polytypic type `Map`, which describes the type of the operations `map` performs at the elements of a structure, is $\Lambda A\ B\ .\ A \to B$. When we specialize `map` to a specific datatype, we will need an instance of this operation for each of the arguments of that datatype. Hence if the datatype has *nv* parameters, we will need *nv* copies of this operation, each of which will need *np* type arguments. To keep track of all of these types, we construct an environment *ets* : `envts` of the form

$$((A_1, B_1, \ldots), (A_2, B_2, \ldots), \ldots, (A_{np}, B_{np}, \ldots))$$

The floor operation $\lfloor T\rfloor_i$ replaces each free variable in T (each argument of the datatype) by the ith variable associated with it by extracting the ith tuple from the environment and then decoding T using this tuple as the type environment (Section 2.3.3).

Returning to our example, for every $\Lambda A\ .\ \cdots$ we encounter during term specialization we will add the elements of the tuple $(A_1, \ldots, A_{np})$ to the front of the tuples already in *ets* (Section 2.6.4). The type of the specialization of the *body* of the lambda abstractions in T_{ex} will then be

$$\texttt{Pt}\langle k\rangle\ (\lfloor A + B \times C\rfloor_1, \ldots, \lfloor A + B \times C\rfloor_{np})$$

When we specialize a function to a closed type ($nv = 0$), *ets* is the tuple containing *np* empty tuples (constructed by `ets_tt np`), and $(\lfloor T\rfloor_1, \ldots, \lfloor T\rfloor_{np})$ reduces to $(T, \ldots, T)$. From a user's perspective (who will usually specialize polytypic functions only to closed types), this means that all *np* arguments of a polytypic function will be initialized to the *same* type.

The full definition of type specialization is given in Figure 2.6; `kit` constructs kind-indexed types and `specType'` returns the application of a kind-indexed type to a tuple $(T_1, \ldots, T_{np})$. This tuple is constructed by `replace_fvs`, whose definition is straight-forward and can be found in the Coq sources (Verbruggen, 2009).

2.6 Term Specialization

A polytypic function is fully specified by giving its polytypic type and the cases for all type constants. The cases for all other types can be inferred. Informally, we can define

[4]Hinze (2000b) uses naming conventions to define the floor operator, but of course naming conventions do not work in a formal setting.

the specialization of a polytypic function `pfn` of type `Pt` to a type $T : k$ as

$$\begin{aligned}
&\texttt{pfn}\langle T : k\rangle : \texttt{Pt}\langle k\rangle\ (\lfloor T\rfloor_1, \ldots, \lfloor T\rfloor_{np})\\
&\texttt{pfn}\langle C : k_C\rangle = \textit{(user defined)}\\
&\texttt{pfn}\langle A : k_A\rangle = f_A\\
&\texttt{pfn}\langle \Lambda A\ .\ T : k_1 \to k_2\rangle = \lambda A_1 \ldots A_{np}\ .\ \lambda f_A\ .\ \texttt{pfn}\langle T : k_2\rangle\\
&\texttt{pfn}\langle T\ U : k_2\rangle =\\
&\quad (\texttt{pfn}\langle T : k_1 \to k_2\rangle)\ (\lfloor U\rfloor_1, \ldots, \lfloor U\rfloor_{np})\ (\texttt{pfn}\langle U : k_1\rangle)
\end{aligned}$$

In this section, we will show how to define the equivalent definition in Coq. The type of term specialization is

```
specTerm : ∀ (np : nat) (k : kind) (t : closed_type k)
 (Pt : PolyType np) (pfn : PolyFn Pt), specType t Pt
```

Since `specTerm` returns a term of the type computed by `specType`, the definition of `specTerm` is a formal proof that term specialization returns terms of the required type. The definition of term specialization is shown in Figure 2.7; it relies on a number of auxiliary lemmas which we do not show but will explain below. Again, the full definitions can be found in the Coq sources. In the remainder of this section we will describe each of the clauses in the definition of `specTerm'`.

2.6.1 Constants

The case for type constants seems straight-forward. After all, we should simply use the definition given by the user. But there is a subtlety we must deal with. Consider the case for the product constant (`tprod`). As part of the definition of the polytypic function, the user will have provided a function `pprod` of type (Section 2.4.2)

```
pprod : specType tprod Pt
```

Recall from Figure 2.4 that `tprod` is syntactic sugar for

```
tconst 0 tt tc_prod
```

As described in Section 2.3, instances of `type` encode kinding derivations; `tprod` encodes the derivation in the empty environment (`tt`)

$$\frac{}{\emptyset \vdash \texttt{tconst tc\_prod} : \star \to \star \to \star}\ \textsc{Const}$$

When `tc_prod` is used inside another type, however, it may well be used in an environment where there *are* free variables. This arises, for instance, in the use of `tc_prod` in the definition of `fork` in Section 2.3.4, where instead we have a derivation of the form

$$\frac{}{A : \star \vdash \texttt{tconst tc\_prod} : \star \to \star \to \star}\ \textsc{Const}$$

```
(* Specialize the polytypic function pfn to type t *)
Fixpoint specTerm' (np nv : nat) (ek : envk nv) (k : kind)
  (t : type nv ek k) (Pt : PolyType np) (pfn : PolyFn Pt) {struct t}
  : ∀ (ets : envts np nv ek) (ef : envf nv ek Pt ets),
     specType' t Pt ets :=
 match t in type nv ek k
 return ∀ (ets : envts np nv ek),
  envf nv ek Pt ets → specType' t Pt ets
 with
 | tconst nv ek k tc => fun ets ef =>
    match tc return specType' (tconst tc) Pt ets with
    | tc_unit => convertS convert_tconst_specTerm (punit pfn)
    | tc_int  => convertS convert_tconst_specTerm (pint pfn)
    | tc_prod => convertS convert_tconst_specTerm (pprod pfn)
    | tc_sum  => convertS convert_tconst_specTerm (psum pfn)
    end
 | tvar nv ek i =>
    fun ets ef => convertT ith_index_f (ggetS i ef)
 | tapp nv ek k1 k2 t1 t2 => fun ets ef =>
    convertS convert_tapp_specTerm
     ((instantiate_tuple (replace_fvs t2 ets)
       (specTerm' t1 pfn ets ef)) (specTerm' t2 pfn ets ef))
 | tlam nv ek k1 k2 t' =>
    fun ets ef => dep_curry
     (fun As => kit k1 Pt As →
                kit k2 Pt
                 (apply_tupleT (replace_fvs (tlam t') ets) As))
     (fun As : tupleT (decK k1) np =>
      (fun fa : kit k1 Pt As =>
       (convertS (convert_tlam_specTerm _ _ _ _)
        (specTerm' t' pfn (add_to_ets As ets) (add_to_ef fa ef)))))
 end.

(* Term specialization for closed types *)
Definition specTerm (np : nat) (k : kind) (t : closed_type k)
  (Pt : PolyType np) (pfn : PolyFn Pt) : specType t Pt :=
 specTerm' t pfn (ets_tt np) tt.
```

Figure 2.7: Term Specialization

In general, we need a function of type

```
specType' (tconst nv ek tc_prod) Pt ets
```

for some number of free variables *nv* and associated kind environment *ek* (*ets* is the environment we need for the type arguments in the generated type, and will be discussed later). We could generalize the definition of the polytypic function given in Section 2.4.2 to

```
Record PolyFn (np : nat) (Pt : PolyType np) : Type :=
 polyFn {
  ...
  pprod : ∀ (nv : nat) (ek : envk nv) (ets : envts np nv ek),
           specType' (tconst nv ek tc_prod) Pt ets ;
  ...
}.
```

However, this complicates both the definition of a polytypic function and the instances the user must provide. Fortunately, it turns out that a polytypic type specialized to `tconst nv ek tc_prod` is the same as that type specialized to `tconst 0 tt tc_prod`, as proven by the following weakening lemma:

Lemma 1 (`convert_tconst_specTerm`)

```
∀ nv ek tc Pt ets,
  specType (tconst 0 tt tc) Pt
= specType' (tconst nv ek tc) Pt ets
```

Proof. Unfolding definitions (Figure 2.6), we find that we have to prove

```
(⌊tconst 0 tt tc⌋₁, ...) = (⌊tconst nv ek tc⌋₁, ...)
```

We can prove that the elements of the tuples are equal: decoding a type constant is independent of the environment provided. We can complete the proof by induction on the length of the tuples (*np*). □

2.6.2 Variables

Recall from the informal definition of term specialization at the start of Section 2.6 that in the case for variables we return the function f_A constructed in the clause for lambda abstraction. However, this informal definition relies on naming conventions which do not translate to a formal setting. Instead we need an environment *ef* containing the appropriate function for each free variable.

The tricky part is to assign a type `envf` to *ef*, since each element in *ef* has a different type. We can compute `envf` using the heterogeneous tuple defined in Section 2.2.3 as follows[5]

[5] `gtupleS` is a variation on `gtupleT` that returns a `Set` rather than a `Type`.

```
Definition envf np nv ek Pt ets :=
 gtupleS (fun i => specType' (tvar nv ek i) Pt ets)
         (elements_of_index nv).
```

The type of the ith function is the specialized type of the ith free variable. Thus, we map `specType` across the tuple containing all possible indices of type `index nv` constructed by `elements_of_index`. Given *ef* we can simply return the ith element in *ef* as the specialized term for variable i.[6] The *construction* of *ef* will be considered in the case for lambda abstraction (Section 2.6.4).

2.6.3 Application

To specialize a polytypic function `pfn` of type `Pt` to a type application $(T\ U)$ we first specialize to $T : k_1 \to k_2$, which will create a term of the form

$$\texttt{specTerm' T pfn ets ef} : \forall\ \texttt{A}_1 \ldots \texttt{A}_{np}, \\ \texttt{kit k1 Pt } (\texttt{A}_1,\ldots,\texttt{A}_{np}) \to \texttt{kit k2 Pt } (\lfloor \texttt{T} \rfloor_1\ \texttt{A}_1,\ \ldots,\ \lfloor \texttt{T} \rfloor_{np}\ \texttt{A}_{np})$$

We instantiate the type variables $\texttt{A}_1 \ldots \texttt{A}_{np}$ in `specTerm' T pfn ets ef` to the elements of the tuple $(\lfloor \texttt{U} \rfloor_1, \ldots, \lfloor \texttt{U} \rfloor_{np})$ using the following function:

```
instantiate_tuple (A : Type) (n : nat) :
 ∀ (args : tupleT A n) (X : tupleT A n → Set),
  quantify_tuple X → X args
```

(see Coq source for a full definition). This leaves us with the following term

$$(\texttt{specTerm' T pfn ets ef})\ \lfloor \texttt{U} \rfloor_1 \ldots \lfloor \texttt{U} \rfloor_{np} \\ : \texttt{kit k1 Pt } (\lfloor \texttt{U} \rfloor_1,\ldots,\lfloor \texttt{U} \rfloor_{np}) \to \texttt{kit k2 Pt } (\lfloor \texttt{T} \rfloor_1\ \lfloor \texttt{U} \rfloor_1,\ldots,\lfloor \texttt{T} \rfloor_{np}\ \lfloor \texttt{U} \rfloor_{np})$$

We can apply this to the polytypic function specialized to the type U, which serendipitously has exactly the right type, and get a term of type

$$\texttt{kit k2 Pt } (\lfloor \texttt{T} \rfloor_1\ \lfloor \texttt{U} \rfloor_1,\ldots,\lfloor \texttt{T} \rfloor_{np}\ \lfloor \texttt{U} \rfloor_{np})$$

Since we are specializing to the application $(T\ U)$, the return type we expect here is

```
specType' (tapp T U) Pt ets
```

We can use the following lemma to complete the definition for application

Lemma 2 (`convert_tapp_specTerm`)

$$\forall\ np\ k1\ k2\ Pt\ ets\ (T : k1 \to k2)\ (U : k1), \\ kit\ k2\ Pt\ (\lfloor T \rfloor_1\ \lfloor U \rfloor_1,\ldots,\lfloor T \rfloor_{np}\ \lfloor U \rfloor_{np}) \\ = specType'\ (tapp\ T\ U)\ Pt\ ets$$

[6]Due to the way we calculate `envf`, we do need one technical lemma (`ith_index_f`) which says that applying a function to the ith element of `elements_of_index` is the same as applying that function to i itself.

Proof. Unfolding definitions (Figure 2.6), we find that we have to prove that

$$(\lfloor \mathtt{T} \rfloor_1\ \lfloor \mathtt{U} \rfloor_1, \ldots, \lfloor \mathtt{T} \rfloor_{np}\ \lfloor \mathtt{U} \rfloor_{np}) = (\lfloor \mathtt{T\ U} \rfloor_1, \ldots, \lfloor \mathtt{T\ U} \rfloor_{np})$$

We can prove that the elements of the tuples are equal: replacing free variables before or after application gives the same result. We can complete the proof by induction on the length of the tuples. □

2.6.4 Lambda Abstraction

In this section we will look at the specialization of a polytypic function `pfn` of type `Pt` to a lambda abstraction ($\Lambda A\ .\ T$). The type of this specialization must be

```
specTerm' (tlam T) pfn ets ef
 : specType' (tlam T) Pt ets
```

which can unfolded to

```
∀ A₁...Aₙₚ, kit k1 Pt (⌊tlam T⌋₁,...,⌊tlam T⌋ₙₚ) →
           kit k2 Pt (⌊tlam T⌋₁ A₁,...,⌊tlam T⌋ₙₚ Aₙₚ)
```

We can construct this term in two steps. We use the specialization of `pfn` to T to construct the body of the expression and use currying to get arguments of the correct type.

Dependent currying

We can construct the required term by first defining a function of the form

```
fun (A₁,...,Aₙₚ) f_A => ...
```

which we can curry to get

```
fun A₁...Aₙₚ f_A => ...
```

Currying this function is, however, not entirely straight-forward. The type of the body of this function

```
kit k2 Pt (⌊tlam T⌋₁ A₁,...,⌊tlam T⌋ₙₚ Aₙₚ)
```

depends on the actual argument tuple that is supplied. We therefore need a *dependent* curry function, which can be defined as

```
Fixpoint dep_curry (A : Type) (n : nat)
  : ∀ (C : tupleT A n → Set) (f : ∀ (x : tupleT A n), C x),
     quantify_tuple C :=
 match n
 return ∀ (C : tupleT A n → Set)
          (f : ∀ (x : tupleT A n), C x), quantify_tuple C
 with
 | O => fun _ f => f tt
 | S m => fun c f a =>
    dep_curry A m (fun args => c (a, args))
```

```
                        (fun args => f (a, args))
    end.
```

Specialization to T

To construct the body of the result, we use the specialization of `pfn` to T:

```
  specTerm' T pfn (add_to_ets (A1,...,Anp) ets) (add_to_ef fA ef)
   : specType' T Pt (add_to_ets (A1,...,Anp) ets)
```

This does not have the correct type, so we need the following conversion lemma:

Lemma 3 (`convert_tlam_specTerm`)

$$\begin{aligned}&\forall\ k2\ T\ Pt\ ets\ (A_1,\ldots,A_{np}),\\ &\quad specType'\ T\ Pt\ (add\_to\_ets\ (A_1,\ldots,A_{np})\ ets)\\ &= kit\ k2\ Pt\ (\lfloor tlam\ T\rfloor_1\,A_1,\ldots,\lfloor tlam\ T\rfloor_{np}\,A_{np})\end{aligned}$$

Proof. Unfolding definitions (Figure 2.6) we find that we have to prove that

$$\begin{aligned}&(\lfloor \mathtt{T}\rfloor_1,\ldots,\lfloor \mathtt{T}\rfloor_{np}) \quad \text{using}\ (\mathtt{add\_to\_ets}\ (\mathtt{A}_1,\ldots,\mathtt{A}_{np})\ \mathtt{ets})\\ =\ &(\lfloor \mathtt{tlam\ T}\rfloor_1\,\mathtt{A}_1,\ldots,\lfloor \mathtt{tlam\ T}\rfloor_{np}\,\mathtt{A}_{np}) \quad \text{using}\ \mathtt{ets}\end{aligned}$$

Again, we can prove that the elements of the tuples are equal: decoding a lambda abstraction and then applying it to A is the same as decoding the body of the lambda abstraction with A added to the front of the type environment. We can complete the proof by induction on the length of the tuples. □

Adding f_A to the function environment

Another difficulty in constructing the specialized term for lambda abstraction is in adding the function f_a to the environment *ef*. The existing environment *ef* has an entry for each free variable in `tlam` T, but variable i in the lambda abstraction will become variable `Some` i in the body T of the lambda abstraction.

Therefore the function f_X associated with the ith variable X in the old environment:

```
  fX : specType' (tvar nv ek i) Pt ets
```

should have type

```
  fX : specType' (tvar (S nv) (k1, ek) (Some i)) Pt
         (add_to_ets (A1,...,Anp) ets)
```

in the new environment. When every function in ef has been shifted in this way, we can add the new function f_A to the start of ef. The following lemma proves that the two types for f_X above are indeed equal:

Lemma 4 (`convert_envf`)

```
∀ nv ek k1 i Pt ets (A_1,...,A_np),
  specType' (tvar nv ek i) Pt ets
= specType' (tvar (S nv) (k1, ek) (Some i)) Pt
   (add_to_ets (A_1,...,A_np) ets)
```

Proof. Unfolding definitions (Figure 2.6), we find that we have to prove

$$\begin{aligned} & (\lfloor \texttt{tvar nv ek i} \rfloor_1, \ldots) \quad \text{using } \texttt{ets} \\ = \; & (\lfloor \texttt{tvar (S nv) (k1, ek) (Some i)} \rfloor_1, \ldots) \\ & \quad \text{using } (\texttt{add\_to\_ets } (\texttt{A}_1, \ldots, \texttt{A}_{np}) \texttt{ ets}) \end{aligned}$$

We can prove that the elements of the tuples are equal: to decode variable i we take the ith element from the environment; this is the same as taking the element Some i from an environment containing one extra element. We can complete the proof by induction on the length of the tuple. □

In addition to shifting each function already in the environment, we also need to convert the type of the function we want to add:

Lemma 5 (convert_envf_elem)

```
∀ nv ek k1 Pt ets (A_1,...,A_np),
  kit k1 Pt (A_1,...,A_np)
= kit k1 Pt (⌊tvar (S nv) (k1, ek) None⌋_1,...)
```

Proof. This reduces to a proof of

$$\begin{aligned} & (\texttt{A}_1, \ldots, \texttt{A}_{np}) \\ = \; & (\lfloor \texttt{tvar (S nv) (k1, ek) None} \rfloor_1, \ldots) \\ & \quad \text{using } (\texttt{add\_to\_ets } (\texttt{A}_1, \ldots, \texttt{A}_{np}) \texttt{ ets}) \end{aligned}$$

Which is trivially true, because decoding the first variable (None) takes the first element from the type environment, which will always be an element from the tuple of A's. We can complete this proof by induction on np. □

2.7 Examples of Polytypic Types and Functions

To conclude this section on polytypic functions and types, we will consider a number of additional polytypic functions.

2.7.1 Equality

We have already discussed the polytypic equality function in Section 1.1.5, and here we will give its formal definition in Coq. This function compares any two elements of the same type for equality, so its polytypic type can be given as:

```
Definition Compare : PolyType 1 :=
 polyType 1 (fun A => A → A → bool).
```

Specializing this type to the datatype `fork` (defined in Section 2.3.4) of kind $\star \to \star$ gives us:

```
Eval compute in specType fork Compare.
 = ∀ A : Set, (A → A → bool) → A × A → A × A → bool
```

The polytypic equality function itself is defined using the `PolyFn` record type:

```
Definition equal : PolyFn Compare :=
 polyFn Compare
  (fun x y => true)
  Zeq_bool
  (fun (A : Set) (f : A → A → bool)
       (B : Set) (g : B → B → bool)
       (x : A × B) (y : A × B) =>
    let (a , b ) := x in
    let (a', b') := y in
     f a a' && g b b')
  (fun (A : Set) (f : A → A → bool)
       (B : Set) (g : B → B → bool)
       (x : A + B) (y : A + B) =>
    match (x, y) with
    | (inl a, inl a') => f a a'
    | (inr b, inr b') => g b b'
    | otherwise => false
    end).
```

which can also be specialized to give us the equality function for type `fork`:

```
Eval compute in specTerm fork equal.
 = fun (A : Set) (f : A → A → bool) (x y : A × A) =>
    let (a1,  a2 ) := x in
    let (a1', a2') := y in
     (f a1 a1') && (f a2 a2')
```

2.7.2 Less Than

A variation on the polytypic equality function is given by `less_than`, below. Like equality, `less_than` will return false when its two arguments have a different structure, but unlike equality it uses a less-than operator to compare integers. One can think of this function as "pairwise comparison".[7]

This comparison function shares its polytypic type with the polytypic equality function: `Compare` (defined above). It can be defined as:

[7]Lexicographical ordering would perhaps be more sensible on arbitrary data structures, but is a more involved example. We use structural comparison instead to avoid getting bogged down in the details of the example.

```
Definition less_than : PolyFn Compare :=
 polyFn Compare
  (fun x y => false)
  (fun x y => Zlt_bool x y)
  (fun (A : Set) (f : A → A → bool)
       (B : Set) (g : B → B → bool)
       (x : A * B) (y : A * B) =>
   let (a, b) := x in let (a', b') := y in
    f a a' && g b b')
  (fun (A : Set) (f : A → A → bool)
       (B : Set) (g : B → B → bool)
       (x : A + B) (y : A + B) =>
   match (x, y) with
   | (inl a, inl a') => f a a'
   | (inr b, inr b') => g b b'
   | otherwise => false
   end).
```

We can specialize this polytypic function to the type `fork` again to get:

```
Eval compute in specTerm fork less_than.
 = fun (A : Set) (f : A → A → bool) (x y : A × A) =>
    let (a1 , a2 ) := x in
    let (a1', a2') := y in
    (f a1 a1') && (f a2 a2')
```

In Section 3.6 we will show that even though `less_than` and `equal` have the same type, they do have different properties. We will prove that equality is commutative but `less_than` is not.

2.7.3 Count

As a second example, we will consider another well-known polytypic function: `count`. This function counts the elements in a data structure, returning a natural number. Note that types of kind $\star$ never contain any elements, so we can simply return 0 when counting units or integers. The polytypic type `Count` and the corresponding polytypic function are defined as:

```
Definition Count : PolyType 1 :=
 polyType 1 (fun A => A → nat).

Definition count : PolyFn Count :=
 polyFn Count
  (fun u => 0)
  (fun z => 0)
  (fun (A : Set) (f : A → nat)
       (B : Set) (g : B → nat)
       (x : A × B) =>
```

```
      let (a, b) := x in
        (f a + g b)%nat)
    (fun (A : Set) (f : A → nat)
         (B : Set) (g : B → nat)
         (x : A + B) =>
      match x with
      | inl a => f a
      | inr b => g b
      end).
```

Again, we can specialize both `Count` and `count` to the type `fork`:

```
Eval compute in specType fork Count.
 = ∀ A : Set, (A → nat) → A × A → nat

Eval compute in specTerm fork count.
 = fun (A : Set) (f : A → nat) (x : A × A) =>
     let (a, a') := x in
      f a + f a'
```

2.7.4 Zero

The polytypic function `zero` sets all integers in a data structure to the value 0. Its type is given by `Zero`, and it is defined as follows:

```
Definition Zero : PolyType 1 :=
 polyType 1 (fun A => A → A).
\end{lstlist

Definition zero : PolyFn Zero :=
 polyFn Zero
   (fun u => u)
   (fun z => 0%Z)
   (fun (A : Set) (f : A → A)
        (B : Set) (g : B → B)
        (x : A * B) =>
     let (a, b) := x in
       (f a, g b))
   (fun (A : Set) (f : A → A)
        (B : Set) (g : B → B)
        (x : A + B) =>
     match x with
     | inl a => inl _ (f a)
     | inr b => inr _ (g b)
     end).
```

Specializing the polytypic type `Zero` to the datatype `fork` gives us the type:

```
Eval compute in specType fork Zero.
```

```
  = ∀ (A : Set), (A → A) → A × A → A × A
```

The specialization of the function `zero` to `fork` looks like:

```
Eval compute in specTerm fork zero.
 = fun (A : Set) (f : A → A) (x : A × A) =>
    let (a, a') := x in
     (f a, f a')
```

2.8 Higher-order Polytypic Functions

By reifying the notion of a polytypic function as a (record) type within Coq, we have made them first-class citizens. They can be passed as arguments to other functions and be computed as results from other functions. We can call such functions *higher-order* polytypic functions or, alternatively, polytypic *combinators*.

To demonstrate this, we will define a combinator which computes the "disjunction" of two other polytypic functions, both of which must have polytypic type `Compare`. We can then use this combinator to define a polytypic less-than-or-equal-to function, given the polytypic functions `less_than` and `equal` which we defined above.

The combinator can be defined as follows:

```
Definition or_poly (pfn1 pfn2 : PolyFn Compare)
  : PolyFn Compare :=
 polyFn Compare
  (fun u v => punit pfn1 u v || punit pfn2 u v)
  (fun i j => pint pfn1 i j || pint pfn2 i j)
  (fun (A : Set) (f : A → A → bool)
       (B : Set) (g : B → B → bool)
       (x : A * B) (y : A * B),
    pprod pfn1 A f B g x y || pprod pfn2 A f B g x y)
  (fun (A : Set) (f : A → A → bool)
       (B : Set) (g : B → B → bool)
       (x : A + B) (y : A + B),
    psum pfn1 A f B g x y || psum pfn2 A f B g x y).
```

Note how it takes two polytypic functions of type `Compare` and computes another polytypic function of the same type. Less-than-or-equal-to is now simply defined as

```
Definition le : PolyFn Compare := or_poly equal less_than.
```

When we specialize this function to $\Lambda A\ .\ \Lambda B\ .\ A + Z \times B$ we get a function which is provably[8] equal to

```
fun (A : Set) (f : A → A → bool)
    (B : Set) (g : B → B → bool)
    (x y : decT tex tt A B) : bool :=
```

[8] See `first_class.v` in the Coq sources.

```
match (x, y) with
| (inl a, inl a’) => f a a’
| (inr (z, b), inr (z’, b’)) => Zle_bool z z’ && g b b’
| _ => false
end
```

Defining such polytypic combinators is non-trivial, but a further discussion falls outside the scope of this thesis.

Chapter 3

Polytypic Properties and Proofs

3.1 Introduction

A key component of polytypic programming is the specialization of kind-indexed types and the specialization of type-indexed programs. In Chapter 2 we demonstrated how type specialization and term specialization can be formalized in the proof assistant Coq (Bertot and Castéran, 2004). As well as an important and obvious stepping stone towards formal proofs about such programs, that chapter also serves as a formal proof that term specialization is correct with respect to type specialization.

In many ways, this chapter can be seen as the Curry-Howard mirror image of Chapter 2. Just like polytypic types are types indexed by a kind, polytypic properties are properties indexed by a kind; and just like polytypic programs are terms indexed by a type, polytypic proofs are proofs indexed by a type. That should come as no surprise, since Curry and Howard tell us that we can read "type" for "property" and "program" for "proof". Nevertheless, the structure of polytypic properties and proofs (interpreted as types and programs or not) is sufficiently different from the structure of types and programs that it introduces many new difficulties that need to be overcome in order to formalize polytypic proofs.

The purpose of this chapter is to describe these difficulties and present their solutions. We make the following contributions:

- Although the formal definition of type and term specialization that we have given in the Chapter 2 makes it theoretically possible to do machine verified proofs over polytypic programs, in reality this is almost impossible without further supporting infrastructure. We provide this infrastructure in the current chapter, so that formal proofs over polytypic programs can be done with very little effort (we give some examples in Section 3.2.3).
- This chapter can be seen as a formal proof that
  - property specialization, the process of specializing a polytypic property to a particular kind, yields well-formed properties (Section 3.4), and that

  - proof specialization, the process of specializing a polytypic proof to a particular type, is correct with respect to property specialization (Section 3.5).

- Seen in another light, it is a formal proof that to do proofs over polytypic programs it suffices to give the instances of the proof for the type constants—just like it suffices to give the instances of a function for the type constants when defining a polytypic function.

In the last chapter we have seen how we can formally interpret the informal notation $\mathit{pfn}\langle T\rangle$ for the specialization of a polytypic function *pfn* to a datatype T and the notation $Pt\langle k\rangle$ for the specialization of a polytypic type Pt to a kind k. We have also seen how we represent the type universe within Coq. To aid readability, we will now feel free to switch back to the informal notation and trust that the reader will understand the interpretation of the informal notation as explained in the previous chapter.

This chapter is an expanded version of (Verbruggen et al., 2009), and the Coq sources associated with the formalization described in this chapter can be found online (Verbruggen, 2009).

3.2 Polytypic Properties and Proofs

In category theory, a functor can be described as a mapping between categories. A functor $F : \mathcal{C} \to \mathcal{D}$ maps every object $C \in \mathcal{C}$ to an object $F(C) \in \mathcal{D}$, and every arrow $f : A \to B \in \mathcal{C}$ to an arrow $F(f) : F(A) \to F(B) \in \mathcal{D}$. In addition, each functor must preserve the two functor laws—preservation of identities and composition:

$$F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$$
$$F(f \circ g) = F(f) \circ F(g)$$

The specialization of `map` to a datatype T, together with that datatype, forms a functor: the datatype T gives us the mapping on objects and `map`$\langle T\rangle$ provides the mapping on arrows. It remains to show that the two functor laws hold for `map`.

In the case of a unary type constructor, such as `fork` $: \star \to \star$, the functor laws take the form:

$$\mathrm{map}\langle\mathrm{fork}\rangle\ \mathrm{id} = \mathrm{id}$$
$$\mathrm{map}\langle\mathrm{fork}\rangle\ (f \circ g) = \mathrm{map}\langle\mathrm{fork}\rangle\ f \circ \mathrm{map}\langle\mathrm{fork}\rangle\ g$$

However, given a type constructor of two arguments such as `maybe_prod` $: \star \to \star \to \star$, the functor laws take a different shape:

$$\mathrm{map}\langle\mathrm{maybe\_prod}\rangle\ \mathrm{id}\ \mathrm{id} = \mathrm{id}$$
$$\mathrm{map}\langle\mathrm{maybe\_prod}\rangle\ (f \circ g)\ (h \circ k) = \mathrm{map}\langle\mathrm{maybe\_prod}\rangle\ f\ h \circ \mathrm{map}\langle\mathrm{maybe\_prod}\rangle\ g\ k$$

It turns out that the shape of these properties depends on the kind of the datatype we specialize to. Fortunately, it turns out that we can state and prove such properties in much the same way as we state the types of polytypic functions and give their implementations, i.e. by making them kind-indexed. In the following sections, we will first give a high level description of how polytypic properties can be stated, and then discuss how this can be formalized in Coq. Section 3.2.3 describes polytypic proofs and finally, Section 3.2.4 discusses some arguably simpler ways we considered for formalizing polytypic properties, and why none of them were appropriate.

3.2.1 Stating Polytypic Properties

To specify a polytypic property we have to give the types of the functions that the property ranges over and the property itself. Take the example that map preserves identity. This property ranges over functions of type `Map`; since `Map` is kind-indexed, it follows that the property itself is kind-indexed:

$$\mathtt{Id}\langle k\rangle\ T : \mathtt{Map}\langle k\rangle\ T\ T \to \mathtt{Prop}$$

In the case for kind $\star$ the type $\mathtt{Map}\langle\star\rangle\ T\ T$ specializes to the function type $T \to T$, and the corresponding definition of the property is:

$$\begin{aligned}
&\mathtt{Id}\langle\star\rangle\ T : (T \to T) \to \mathtt{Prop}\\
&\mathtt{Id}\langle\star\rangle\ T = \lambda f : T \to T\ .\ \forall x : T\ .\ f\ x = x
\end{aligned}$$

To prove that this property (for kind $\star$) holds for the polytypic map function specialized to a type T, we must prove that the property holds for $f = \mathtt{map}\langle T\rangle$, i.e.

$$\forall x : T\ .\ \mathtt{map}\langle T\rangle\ x = x$$

In other words: in the case for kind $\star$ we have to prove that $\mathtt{map}\langle T\rangle$ is *itself* the identity function.

From the definition of the type of the property and the case for kind $\star$, we can derive the property for other kinds. For example, the instance for kind $\star \to \star$ will be:

$$\begin{aligned}
&\mathtt{Id}\langle\star\to\star\rangle\ T : (\forall A_1\ A_2 : \star\ .\ (A_1 \to A_2) \to T\ A_1 \to T\ A_2) \to \mathtt{Prop}\\
&\mathtt{Id}\langle\star\to\star\rangle\ T = \lambda(f : \forall A_1\ A_2 : \star\ .\ (A_1 \to A_2) \to T\ A_1 \to T\ A_2)\ .\\
&\qquad \forall (A : \star)\ (g : A \to A)\ .\ \mathtt{Id}\langle\star\rangle\ A\ g \to \mathtt{Id}\langle\star\rangle\ (T\ A)\ (f\ A\ A\ g)\\
&\equiv \lambda f\ .\ \forall (A : \star)\ (g : A \to A)\ .\\
&\qquad (\forall y : A\ .\ g\ y = y) \to \forall x : T\ A\ .\ f\ A\ A\ g\ x = x
\end{aligned}$$

Instantiating f by $\mathtt{map}\langle T\rangle$ gives the property we would expect:

$$\begin{aligned}&\mathtt{Id}\langle\star\rightarrow\star\rangle\ T\ \mathtt{map}\langle T\rangle =\\ &\quad\forall(A:\star)\ (g:A\rightarrow A)\ .\ (\forall y:A\ .\ g\ y=y)\rightarrow\forall x:T\ A\ .\ \mathtt{map}\langle T\rangle\ A\ A\ g\ x=x\end{aligned}$$

Given a type $A:\star$ and a function $g:A\rightarrow A$ such that g is the identity function on A, we must show that the property holds for $\mathtt{map}\langle T\rangle\ A\ A\ g$. Rephrased, we have to prove that given an identity function g, mapping g across an element of type T is also an identity function.

The property that map preserves composition is more complicated: composition ranges over *three* functions of type `Map`, each instantiated at a different type:

$$\mathtt{Comp}\langle k\rangle\ T_1\ T_2\ T_3 : \mathtt{Map}\langle k\rangle\ T_2\ T_3\times\mathtt{Map}\langle k\rangle\ T_1\ T_2\times\mathtt{Map}\langle k\rangle\ T_1\ T_3\rightarrow\mathtt{Prop}$$

In the case for kind $\star$ the type $\mathtt{Map}\langle\star\rangle\ T_1\ T_2$ specializes to the function type $T_1\rightarrow T_2$, and the property is defined as:

$$\begin{aligned}&\mathtt{Comp}\langle\star\rangle\ T_1\ T_2\ T_3 : (T_2\rightarrow T_3)\times(T_1\rightarrow T_2)\times(T_1\rightarrow T_3)\rightarrow\mathtt{Prop}\\ &\mathtt{Comp}\langle\star\rangle\ T_1\ T_2\ T_3 = \lambda(f_1,f_2,f_3)\ .\ \forall x:T_1\ .\ f_1\ (f_2\ x)=f_3\ x\end{aligned}$$

As before, the definition of the property for other kinds can now be derived. For example, the instance for kind $\star\rightarrow\star$ is:

$$\begin{aligned}&\mathtt{Comp}\langle\star\rightarrow\star\rangle\ T_1\ T_2\ T_3 :\\ &\quad\mathtt{Map}\langle\star\rightarrow\star\rangle\ T_2\ T_3\times\mathtt{Map}\langle\star\rightarrow\star\rangle\ T_1\ T_2\times\mathtt{Map}\langle\star\rightarrow\star\rangle\ T_1\ T_3\rightarrow\mathtt{Prop}\\ &\mathtt{Comp}\langle\star\rightarrow\star\rangle\ T_1\ T_2\ T_3 = \lambda(f_1,f_2,f_3)\ .\ \forall A_1\ A_2\ A_3\ (g_1,g_2,g_3)\ .\\ &\quad\mathtt{Comp}\langle\star\rangle\ A_1\ A_2\ A_3\ (g_1,g_2,g_3)\rightarrow\\ &\quad\mathtt{Comp}\langle\star\rangle\ (T_1\ A_1)\ (T_2\ A_2)\ (T_3\ A_3)\ (f_1\ A_2\ A_3\ g_1,f_2\ A_1\ A_2\ g_2,f_3\ A_1\ A_3\ g_3)\\ &\ \equiv\lambda(f_1,f_2,f_3)\ .\ \forall A_1\ A_2\ A_3\ (g_1,g_2,g_3)\ .\ (\forall y:A_1\ .\ g_1\ (g_2\ y)=g_3\ y)\rightarrow\\ &\qquad\forall x:T1\ A1\ .\ f_1\ A_2\ A_3\ g_1\ (f_2\ A_1\ A_2\ g_2\ x)=f_3\ A_1\ A_3\ g_3\ x\end{aligned}$$

The property for map will then be

$$\begin{aligned}&\mathtt{Comp}\langle\star\rightarrow\star\rangle\ T\ T\ T\ (\mathtt{map}\langle T\rangle,\mathtt{map}\langle T\rangle,\mathtt{map}\langle T\rangle) =\\ &\quad\forall A_1\ A_2\ A_3\ (g_1,g_2,g_3)\ .\ (\forall y:A_1\ .\ g_1\ (g_2\ y)=g_3\ y)\rightarrow\\ &\quad\forall x:T\ A_1\ .\ \mathtt{map}\langle T\rangle\ A_2\ A_3\ g_1\ (\mathtt{map}\langle T\rangle\ A_1\ A_2\ g_2\ x)=\mathtt{map}\langle T\rangle\ A_1\ A_3\ g_3\ x\end{aligned}$$

This is a generalization of the usual property, which we can obtain by instantiating g_3 by $g_1\circ g_2$.

3.2.2 Polytypic Properties, Formally

We define a polytypic property using the following record type:

```
Record PolyProp (nt nx np : nat) (Pt : PolyType np):=
 polyProp {
  idxs : tupleT (tupleT (index nt) np) nx;
  propKindStar : ∀ (types : tupleT (decK star) nt),
   gtupleTS (kit star Pt) (reindex_tuple idxs types) → Prop
 }.
```

The record contains two fields: the first (`idxs`, described in more detail below) gives information about the type of the property, and the second (`propKindStar`) gives the property for kind $\star$.

The record is dependent on four arguments:

		Id	Comp
`nt`	number of type arguments of the property	1	3
`nx`	number of function arguments of the property	1	3
`np`	number of type arguments of the polytypic type	2	2
`Pt`	polytypic type the property ranges over	`Map`	`Map`

Given nt type arguments $T_1 \ldots T_{nt}$, the type of a polytypic property indexed by a kind k generally looks like[1]

$$\mathtt{Pt}\langle k\rangle\ \underbrace{(T_1,\ldots,T_{nt})}_{1} \times \cdots \times \mathtt{Pt}\langle k\rangle\ \underbrace{(T_1,\ldots,T_{nt})}_{nx} \rightarrow \mathtt{Prop}$$

where $\underbrace{(T_1,\ldots,T_{nt})}_{i}$ takes the correct np type arguments for the ith occurrence of `Pt` from the tuple of type arguments $(T_1,\ldots,T_{nt})$ associated with the property; e.g., for the case of preservation of composition for map, we have that $\underbrace{(T_1,T_2,T_3)}_{1} = (T_2,T_3)$, $\underbrace{(T_1,T_2,T_3)}_{2} = (T_1,T_2)$ and $\underbrace{(T_1,T_2,T_3)}_{3} = (T_1,T_3)$; compare to the type of `Comp`, above. This mapping is given by `idxs` in the description of the polytypic property.

The property for kind $\star$ is given by `propKindStar`, given the same tuple of type arguments $(T_1,\ldots,T_{nt})$, and a tuple containing nx function arguments:

$$(g_1 : \mathtt{Pt}\langle\star\rangle\ \underbrace{(T_1,\ldots,T_{nt})}_{1}, \ldots, g_{nx} : \mathtt{Pt}\langle\star\rangle\ \underbrace{(T_1,\ldots,T_{nt})}_{nx})$$

Since every element in this second tuple has a different type, the type of the entire tuple is described as a heterogeneous tuple.[2] A heterogeneous tuple `gtupleTS` $f\ (x_1,\ldots,x_n)$ is a tuple of type $(f\ \ x_1 \times \cdots \times f\ \ x_n)$. In this case, the function f that we apply is `kit` $\star$ `Pt`, which is the Coq equivalent of $\mathtt{Pt}\langle\star\rangle$; and the tuple $(x_1,\ldots,x_n)$ that

[1]This limits the expressiveness of polytypic properties, as they can only refer to a single polytypic type. A generalization should not be difficult, but is left as future work.

[2]`gtupleTS` is a particular variety of a heterogeneous tuple; details can be found in the Coq formalization but are not important here. We discuss heterogeneous tuples in more detail in Section 2.2.3.

we supply is the tuple of tuples of types $(\underbrace{(T_1,\ldots,T_{nt})}_{1},\ldots,\underbrace{(T_1,\ldots,T_{nt})}_{nx})$, which is created by `reindex_tuple`.

Hopefully two examples will go a long way towards clarifying these definitions. The property that map preserves identity can be stated using our library in Coq as[3]

```
Definition Id : PolyProp 1 1 Map :=
  polyProp 1 1 Map
    ((1, 1))
    (fun T f => ∀ x : T, f x = x).
```

Note that we only provide three arguments to `PolyProp`: `nt`, `nx` and `Pt`, the argument `np` is implicit in the type of `Map` : `PolyType np` and can therefore be omitted. Similarly, the property that map preserves composition can be stated as

```
Definition Comp : PolyProp 3 3 Map :=
  polyProp 3 3 Map
    ((2, 3), (1, 2), (1, 3))
    (fun (T1, T2, T3) (f1, f2, f3) =>
      ∀ x : T1, f1 (f2 x) = f3 x).
```

3.2.3 Polytypic Proofs

When we define a polytypic (that is, type-indexed) function, it suffices to give the implementation for the type constants; all other cases can be derived. Likewise, in a polytypic proof it suffices to prove the property for the type constants. Our development can be regarded as a formal proof that this is indeed sufficient.

The definition of a polytypic proof mirrors the definition of a polytypic function (Section 2.4.2):

```
Record PolyProof (nt nx np : nat) (Pt : PolyType np)
   (pfn : PolyFn Pt) (Pp : PolyProp nt nx Pt) : Type :=
 polyProof {
  prfUnit : specProp tunit Pp
             (cst_closed tunit pfn (idxs Pp)) ;
  prfInt  : specProp tint Pp
             (cst_closed tint pfn (idxs Pp)) ;
  prfProd : specProp tprod Pp
             (cst_closed tprod pfn (idxs Pp)) ;
  prfSum  : specProp tsum Pp
             (cst_closed tsum pfn (idxs Pp))
}
```

where `cst_closed` generates the tuple of polytypic functions for which we want to prove the property `Pp`, instantiated at the correct types. So to define a polytypic proof, we

[3]We have taken some liberties with notation to keep things simple: we use natural numbers for indices, and assume that we can decompose tuples as part of a function definition. Such syntactic sugar can be added to the Coq library as well, but we have left this to future work for now.

must provide the proofs for the type constants `tunit`, `tint`, `tprod` and `tsum`. Here is an example: the proof that map preserves composition.

```
Lemma map_Comp : PolyProof map Comp.
Proof.
(* This is a polyProof *)
  apply (polyProof map Comp);
(* Solve case for unit and int by auto tactic *)
  compute; auto; intros.
(* Case for products *)
  destruct x ; rewrite H ; rewrite H0 ; auto.
(* Case for sums *)
  destruct x ; [rewrite H | rewrite H0] ; auto.
Defined.
```

Same as for `PolyProp`, the arguments `np`, `nt`, `nx` and `Pt` to `PolyProof` are implicit in the types of `map` and `Comp` and can therefore be omitted. The details of the proof will be obscure to people not familiar with Coq, but they do not matter for our current purposes. Suffice to say that the proof is easy; the cases for unit and int are solved automatically (by the `auto` tactic), and the other cases follow straightforwardly from the appropriate assumptions about the components of the pair or the value in the sum respectively. In Section 6.1.1 we show the case for products in a more human-readable form.

The polytypic proof that map preserves identity is very similar:

```
Definition map_Id : PolyProof map Id.
Proof.
(* This is a polyProof *)
  apply (polyProof map Id);
(* Solve case for unit and int by auto tactic *)
  compute; auto; intros.
(* Case for products *)
  destruct x; rewrite H; rewrite H0; auto.
(* Case for sums *)
  destruct x; [rewrite H | rewrite H0]; auto.
Defined.
```

Because of this similarity it should be possibly to write a Coq tactic (proof search algorithm) to prove many of these polytypic proofs fully automatically; we have left this to future work.

To anticipate the development of proof specialization in Section 3.5, we can now prove that map specialized to T_{ex} preserves composition simply by applying proof specialization to the Lemma `map_Comp`:

`specProof` T_{ex} `map_Comp`

3.2.4 Alternative Definitions

To specify a property using our formalization, the user must specify the type of the property by means of the `idxs` tuple of tuples of indices, and the property for kind $\star$. The mechanism for specifying the type of the property may seem non-obvious. In this section, we give the rationale for choosing this approach; this does not affect the remainder of this chapter and can safely be skipped should the reader wish to do so.

In the definition of a polytypic type (`PolyType`, Section 2.4.1) we do not ask the user to specify the kind of the polytypic type. We do not need to, because we can construct it given *np*: it will always be

$$\underbrace{k \to k \to \cdots \to k}_{np} \to \star$$

That is, given *np* type arguments of kind *k*, we construct a type of kind $\star$.

Unfortunately, the situation is not so simple for properties: as mentioned in Section 3.2.2, the type of a property looks like

$$\mathtt{Pt}\langle k\rangle\ \underbrace{(T_1,\ldots,T_{nt})}_{1} \times \cdots \times \mathtt{Pt}\langle k\rangle\ \underbrace{(T_1,\ldots,T_{nt})}_{nx} \to \mathtt{Prop}$$

where the problem is to find the mapping $\underbrace{(T_1,\ldots,T_{nt})}_{i}$ for each occurrence of `Pt`.

The most obvious solution might seem to simply ask the user to provide the complete type of the property, given the tuple $(T_1,\ldots,T_{nt})$. However, this is far too liberal: specialization relies on a particular shape of the type of the property (see Section 3.4). Intuitively, the more leeway we give to the user, the less we can assume about the type of the property and the more difficult it becomes to derive properties for kinds other than $\star$, much less automate the derivation of the corresponding proofs.

One possible alternative is to ask the user for a tuple of tuples of types, rather than the tuple of tuples of indices `idxs`:

```
fnTypeArgs : ∀ k : kind, tupleT (decK k) nt →
 tupleT (tupleT (decK k) np) nx
```

Temporarily denoting this function by $[\![\cdot]\!]$, during the development of property specialization we need a lemma that says that

$$[\![(T_1,\ldots,T_n)]\!]\ [\![(A_1,\ldots,A_n)]\!] = [\![(T_1\ A_1,\ldots,T_n\ A_n)]\!]$$

In other words, `fnTypeArgs` should only “shuffle” its input arguments. Since this is not true for an arbitrary `fnTypeArgs`, we would have to require it as a separate lemma in the record. We felt it was simpler to ask for the indices and do the shuffling ourselves.

We attempted to avoid the problem altogether by leaving this shuffling to the case for kind $\star$. The type of the property would then become

$$(\forall Ts : k^{np}\ .\ \mathtt{Pt}\langle k\rangle\ Ts) \times \cdots \times (\forall Ts : k^{np}\ .\ \mathtt{Pt}\langle k\rangle\ Ts) \to \mathtt{Prop}$$

where k^n is the tuple of n types of kind k. Again, this does not give us enough information for property specialization. In particular, when specializing the property for kind $k_1 \to k_2$, we need to construct the property for kind k_2 given the property for kind k_1. As part of the property, we need to construct the function arguments to the property; if the function argument for kind $k_1 \to k_2$ is f (e.g., `map`) and the function argument for kind k_1 is x (e.g., the function that we are mapping across the data structure), then the function argument for kind k_2 is $f\ x$. To be able to apply f to x we need to find the right type parameters to instantiate f, and using this approach we do not have this information.

3.3 Reasoning about Equality

One of the important technical difficulties in *term* specialization was to find the appropriate type conversions (the `convert_`*xxx* lemmas in Section 2.6). *Proof* specialization reasons *about* specialized terms and consequently reasoning about these conversion lemmas was one of the major technical difficulties in the formalization of proof specialization. In particular some of the definitions of term specialization had to be adapted to make this reasoning feasible. In this section, we explain some of these difficulties.

The standard definition of equality in Coq only allows to state equality between terms of the same type:

$$\frac{}{(e : T) =_T (e : T)}\ \textsc{Refl}$$

This definition is often too restrictive as it does not allow us to state, much less prove, that $e_1 : T_1$ is equal to $e_2 : T_2$ for two *provably equal* but not *syntactically equal* types T_1 and T_2. Heterogeneous or *John Major* equality (McBride, 2002) is a generalization of the standard equality relation which allows us to state equalities between terms of a different type, even though its only constructor still only allows us to prove equality between terms of the same type:

$$\frac{}{(e : T) \simeq_{T,T} (e : T)}\ \text{JM-}\textsc{Refl}$$

To prove $(e_1 : T_1) \simeq_{T_1,T_2} (e_2 : T_2)$ we must first show that $T_1 = T_2$, then that $e_1 = e_2$, at which point JM-Refl finishes the proof.

Unfortunately, given some property $P : \forall A : \texttt{Set}, A \to \texttt{Prop}$ and $e_1 \simeq_{T_1,T_2} e_2$, proving $P_{T_2}\ e_2$ given $P_{T_1}\ e_1$ is not entirely straightforward: simply replacing e_1 by e_2 in $P_{T_1}\ e_1$ would yield the ill-typed term $P_{T_1}\ e_2$. Instead, the proof usually looks like

$$\begin{array}{ll}
 & P_{T_1}e_1 \rightarrow P_{T_2}e_2 \\
 & \quad \{ \text{ generalize over the proof that } e_1 \simeq_{T_1,T_2} e_2 \ \} \\
\Leftarrow & \forall (pf : e_1 \simeq_{T_1,T_2} e_2), P_{T_1}e_1 \rightarrow P_{T_2}e_2 \\
 & \quad \{ \text{ generalize over } e_1 \ \} \\
\Leftarrow & \forall (x : T_1)(pf : x \simeq_{T_1,T_2} e_2), P_{T_1}x \rightarrow P_{T_2}e_2 \\
 & \quad \{ \text{ replace } T_1 \text{ by } T_2 \ \} \\
= & \forall (x : T_2)(pf : x \simeq_{T_2,T_2} e_2), P_{T_2}x \rightarrow P_{T_2}e_2
\end{array}$$

The final case is easily proven, as we can use pf to replace x by e_2 (which now both have type T_2).

Such a proof is not always as straight-forward, however. First, when the terms get large it is not always obvious which terms need to be generalized over and in which order. Second, suppose we have some dependent type $D : T \rightarrow \texttt{Set}$, and we have a function $f : \forall (t : T), D\ t \rightarrow T'$. Suppose also that we have two elements $t_1, t_2 : T$ and an element $d_1 : D\ t_1$ and $d_2 : D\ t_2$, and that we know that $d_1 \simeq_{D\ t_1, D\ t_2} d_2$ (but $t_1 \neq t_2$). Now, it may be the case that f uses its first argument only to determine the type of the second argument (i.e., that f is parametric in its first argument), in which case we should be able to show that

$$f\ t_1\ d_1 = f\ t_2\ d_2$$

but this will not hold generally for arbitrary f. Depending on the structure of f (and its argument), this may or may not be difficult to prove.

In particular, one common function that we will use in the proofs is

$$\texttt{convert} : \forall A\ B : \texttt{Set}, A = B \rightarrow A \rightarrow B$$

Given an element of type A, this function converts it into an element of type B, provided that we pass in a proof that $A = B$. To aid readability we will assume that the arguments A, B and the proof that $A = B$ are implicit. Associated with this function is a lemma proving that this conversion does not change the actual element, only its type:

Lemma 6 (Convert Identity)

$$\forall (A\ B : \mathit{Set})\ (x : A), A = B \rightarrow x \simeq_{A,B} \texttt{convert}\ p\ x$$

However, even armed with this lemma proofs about heterogeneous equality are often difficult as `convert` x cannot simply be replaced by x (since this would yield ill-formed terms). For example, consider the case where f takes an additional argument i, which it uses to index the vector d_1. Then proving that

$$f\ i\ t_1\ d_1 = f\ i\ t_2\ (\texttt{convert}\ d_1)$$

may be difficult: this needs to be proven as a property of f, but the occurrence of `convert` on the right hand side might make it near impossible to do a proof by induction. In such cases, it is often better to "push down" converts deeper into terms (so that every element of the vector is converted, rather than the entire vector).

Unfortunately, the term specialization of a polytypic function to a particular type contains many calls to `convert`. To consider one (simple) example, recall that our type universe `type` encodes kind derivations rather than the syntax of types. If C is a type constant of kind k, we have that $\emptyset \vdash C : k$—since C does not have any free variables, C has kind k in the empty environment. However, we also have that $\Gamma \vdash C : k$ for all environments Γ; this is known as *weakening*.

When the user defines a polytypic function, they must give the definition of the function for each type constant C, which will have type $\mathtt{Pt}\langle k\rangle\ (\lfloor\emptyset \vdash C : k\rfloor_1, \ldots, \lfloor\emptyset \vdash C : k\rfloor_n)$. Term specialization however is defined over open types, that is, over kind derivations of the form $\Gamma \vdash T : k$ for some type T and kind k.

This is important, because even though the user may only apply term specialization to closed types, term specialization is defined by induction on types; when it encounters an abstraction, it needs to introduce a new type assumption into the environment and the body of the lambda is no longer closed.[4] In Section 2.6.1, we therefore proved that

Lemma 7 (`convert_tconst_specTerm`)

$$\begin{aligned} &\mathtt{Pt}\langle k\rangle\ (\lfloor\emptyset \vdash C : k\rfloor_1, \ldots, \lfloor\emptyset \vdash C : k\rfloor_n) \\ = {}&\mathtt{Pt}\langle k\rangle\ (\lfloor\Gamma \vdash C : k\rfloor_1, \ldots, \lfloor\Gamma \vdash C : k\rfloor_n) \end{aligned}$$

We can prove this lemma by showing that both argument tuples are the same; since type constants contain no free variables, both tuples evaluate to $(C^*, \ldots, C^*)$ where C^* is the Coq type that corresponds to C (the decoding of C). The specialization of a polytypic function for a type constant is then the definition given by the user converted using the Lemma 7:

$$\mathtt{convert}\ (\text{Lemma } 7)\ \textit{(user definition)}$$

During proof specialization we have to prove a similar conversion: when we construct a proof of a property `Pp` for some polytypic function `pfn`, we have to show that

Lemma 8 (`convert_tconst_specProof`)

$$\begin{aligned} &\mathtt{Pp}\langle k\rangle\ (\lfloor\emptyset \vdash C : k\rfloor_0, \ldots)\ (\mathtt{pfn}\langle\emptyset \vdash C : k\rangle, \ldots) \\ = {}&\mathtt{Pp}\langle k\rangle\ (\lfloor\Gamma \vdash C : k\rfloor_0, \ldots)\ (\mathtt{pfn}\langle\Gamma \vdash C : k\rangle, \ldots) \end{aligned}$$

[4] It is not possible to close the body, because the type assumption corresponds to a real Coq datatype, whereas the body of the lambda is a *code* for a type in the universe.

To prove this lemma, we again show that the two argument tuples are the same. We already proved this for the first argument tuples; remains to show that the second argument tuples are identical. Since terms of the form $\texttt{pfn}\langle\emptyset \vdash C : k\rangle$ have type $\texttt{Pt}\langle k\rangle\ (\lfloor\emptyset \vdash C : k\rfloor_1, \ldots, \lfloor\emptyset \vdash C : k\rfloor_{np})$ but terms of the form $\texttt{pfn}\langle\Gamma \vdash C : k\rangle$ have type $\texttt{Pt}\langle k\rangle\ (\lfloor\Gamma \vdash C : k\rfloor_1, \ldots, \lfloor\Gamma \vdash C : k\rfloor_{np})$, we will need to use heterogeneous equality:

Lemma 9

$$\begin{aligned} &\texttt{pfn}\langle\emptyset \vdash C : k\rangle \\ \simeq_{\texttt{Pt}\langle k\rangle\ (\lfloor\emptyset\vdash C:k\rfloor_0,\ldots),\texttt{Pt}\langle k\rangle\ (\lfloor\Gamma\vdash C:k\rfloor_0,\ldots)} &\\ &\texttt{pfn}\langle\Gamma \vdash C : k\rangle \end{aligned}$$

The specialization of a polytypic function to a type constant simply returns the definition that was given by the programmer converted by Lemma 7. Hence, both sides of the equality reduce to

$$\begin{aligned} &\texttt{convert}\ (\text{Lemma 7 at }\emptyset)\ \textit{(user definition)} \\ \simeq_{\texttt{Pt}\langle k\rangle\ (\lfloor\emptyset\vdash C:k\rfloor_0,\ldots),\texttt{Pt}\langle k\rangle\ (\lfloor\Gamma\vdash C:k\rfloor_0,\ldots)} &\\ &\texttt{convert}\ (\text{Lemma 7 at }\Gamma)\ \textit{(user definition)} \end{aligned}$$

which follows from Lemma 6. We can now prove Lemma 8 using the method that we sketched above: generalize over Lemma 9, rewrite with Lemma 7, and complete the proof.

Although this was only a simple example, this kind of reasoning about heterogeneous equalities involving converts is very common throughout the proof and far from straightforward.

3.4 Property Specialization

Section 3.2.2 explains the general form of a polytypic property. For a specific property, the user specifies the type of the property and gives the property for kind $\star$; the case for kind $k_1 \to k_2$ can then be derived.

The informal definition of property specialization is very similar to that of type

```
Fixpoint kip (k : kind) (nt nx np : nat) (Pt : PolyType np)
 (Pp : PolyProp nt nx Pt) {struct k}
 : ∀ types : tupleT (decK k) nt,
   gtupleTS (kit k Pt) (reindex_tuple (idxs Pp) types) → Prop :=
 match k
 return ∀ types : tupleT (decK k) nt,
  gtupleTS (kit k Pt) (reindex_tuple (idxs Pp) types) → Prop
 with
 | star => fun types fns => propKindStar Pp types fns
 | karr k1 k2 => fun types fns => quantify_tuple_Prop
    (fun types' : tupleT (decK k1) nt =>
      ∀ fns' : gtupleTS (kit k1 Pt) (reindex_tuple (idxs Pp) types'),
       kip k1 Pp types' fns' →
       kip k2 Pp (apply_tupleT types types') (app_fs fns fns'))
 end.

Definition specProp' (nt nx np nv : nat) (k : kind)
  (ek : envk nv) (t : type nv ek k) (Pt : PolyType np)
  (Pp : PolyProp nt nx Pt) (ets : envts nt nv ek)
  : gtupleTS (kit k Pt)
     (reindex_tuple (idxs Pp) (replace_fvs t ets)))
    →  Prop :=
 kip Pp (replace_fvs t ets).

Definition specProp (nt nx np : nat) (k : kind)
  (t : closed_type k) (Pt : PolyType np) (Pp : PolyProp nt nx Pt)
  : gtupleTS (kit k Pt)
     (reindex_tuple (idxs Pp) (replace_fvs t (ets_tt nt)))
    → Prop :=
 specProp' t Pp (ets_tt nt).
```

Figure 3.1: Property Specialization

specialization (Section 2.5):

$$\begin{aligned}
&\texttt{Pp}\langle k\rangle\ T_1 \ldots T_{nt} : \texttt{Pt}\langle k\rangle\ \underbrace{(T_1,\ldots,T_{nt})}_{1} \times \cdots \times \texttt{Pt}\langle k\rangle\ \underbrace{(T_1,\ldots,T_{nt})}_{nx} \rightarrow \texttt{Prop}\\
&\texttt{Pp}\langle \star\rangle\ T_1 \ldots T_{nt} = \textit{(user defined)}\\
&\texttt{Pp}\langle k_1 \rightarrow k_2\rangle\ T_1 \ldots T_{nt} = \lambda(f_1,\ldots,f_{nx})\ .\ \forall A_1 \ldots A_{nt} : k_1\ .\ \forall(g_1,\ldots,g_{nx})\ .\\
&\quad \texttt{Pp}\langle k_1\rangle\ (A_1,\ldots,A_{nt})\ (g_1,\ldots,g_{nx}) \rightarrow\\
&\quad \texttt{Pp}\langle k_2\rangle\ (T_1\ A_1,\ldots,T_{nt}\ A_{nt})\ (\texttt{app\_fs}\ (f_1,\ldots,f_{nx})\ (g_1,\ldots,g_{nx}))
\end{aligned}$$

The Coq formalization of this definition is given as `kip` (kind-indexed property) in Figure 3.1.

When we compare this definition to that of type specialization (Section 2.5), we see that the only significant difference (other than its type) is that the kind-indexed property takes an extra tuple of function arguments $(f_1,\ldots,f_{nx})$. Consider the property of preservation of composition, specialized to kind $\star \rightarrow \star$ (Section 3.2.1):

$$\forall g_1\ g_2\ g_3\ .\ (g_1 \circ g_2) = g_3 \rightarrow f_1\ g_1 \circ f_2\ g_2 = f_3\ g_3$$

In the case that we want to prove this property for `map` specialized to the datatype $\texttt{fork} : \star \rightarrow \star$, $nx = 3$ and (f_1, f_2, f_3) will all be instantiated to $\texttt{map}\langle\texttt{fork}\rangle$, the tuple (g_1, g_2, g_3) corresponds to the three functions in the informal statement of the property, and

$$\texttt{app\_fs}\ (f_1,\ldots,f_{nx})\ (g_1,\ldots,g_{nx})$$

corresponds to the application of $\texttt{map}\langle\texttt{fork}\rangle$ to each of (g_1, g_2, g_3). This is not *quite* straight-forward application, however. The types of each f_i and g_i are

$$\begin{aligned}
f_i &: \texttt{Pt}\langle k_1 \rightarrow k_2\rangle\ \underbrace{(T_1,\ldots,T_{nt})}_{i}\\
g_i &: \texttt{Pt}\langle k_1\rangle\ \underbrace{(A_1,\ldots,A_{nt})}_{i}
\end{aligned}$$

From Section 2.5 we know that a polytypic type specialized to an arrow kind $k_1 \rightarrow k_2$ takes the form

$$\forall A_1 \ldots A_{np} : k_1\ .\ \texttt{Pt}\langle k_1\rangle\ (A_1,\ldots,A_{np}) \rightarrow \cdots$$

Hence, we first instantiate $A_1 \ldots A_{np}$ in f_i by $\underbrace{(A_1,\ldots,A_{nt})}_{i}$ to get a term of type

$$\texttt{Pt}\langle k_1\rangle\ \underbrace{(A_1,\ldots,A_{nt})}_{i} \rightarrow \texttt{Pt}\langle k_2\rangle\ \underbrace{(T_1\ A_1,\ldots,T_{nt}\ A_{nt})}_{i}$$

We see that the argument expected here matches the type of g_i exactly, so we apply this to g_i to get a term of type

$$\texttt{Pt}\langle k_2\rangle\ \underbrace{(T_1\ A_1,\ldots,T_{nt}\ A_{nt})}_{i}$$

The function `app_fs` does exactly this: instantiate f_i with the appropriate type arguments and then apply it to g_i (the definition can be found in the Coq sources).

Given $\texttt{Pp}\langle k\rangle$, we can now define property specialization by applying it to the correct type arguments:

$$\texttt{Pp}\langle k\rangle\ (\lfloor T\rfloor_1, \ldots, \lfloor T\rfloor_{nt})$$

This follows type specialization (Section 2.5) exactly. The corresponding Coq definition is given as `specProp'` in Figure 3.1 and like `specType`, `specProp` instantiates `specProp'` to closed types.

3.5 Proof Specialization

Informally, proof specialization can be defined as:

$$\begin{aligned}
&\texttt{prf}\langle T : k\rangle : \texttt{Pp}\langle k\rangle\ (\lfloor T\rfloor_1, \ldots, \lfloor T\rfloor_{nt})\ (\texttt{pfn}\langle T\rangle_1, \ldots, \texttt{pfn}\langle T\rangle_{nx})\\
&\texttt{prf}\langle C : k_C\rangle = \textit{(user defined)}\\
&\texttt{prf}\langle A : k_A\rangle = p_A\\
&\texttt{prf}\langle \Lambda A\ .\ T : k_1 \to k_2\rangle = \lambda A_1 \ldots A_{nt}\ .\ \lambda p_A\ .\ \texttt{prf}\langle T : k_2\rangle\\
&\texttt{prf}\langle T\ U : k_2\rangle =\\
&\quad (\texttt{prf}\langle T : k_1 \to k_2\rangle)\ (\lfloor U\rfloor_1, \ldots, \lfloor U\rfloor_{nt})\ (\texttt{pfn}\langle U\rangle_1, \ldots, \texttt{pfn}\langle U\rangle_{nx})\ (\texttt{prf}\langle U : k_1\rangle)
\end{aligned}$$

This definition is very similar to the definition of term specialization that we gave in Section 2.6, except that proofs need an additional tuple of arguments

$$(\texttt{pfn}\langle T\rangle_1, \ldots, \texttt{pfn}\langle T\rangle_{nx})$$

corresponding to the polytypic functions for which we want to prove the property.

Like the definition of term specialization, this truly is an informal definition: many details are omitted. In particular, since T can be an open type (contain free variables), we need some information about these free variables, which is provided by three environments:

ets For each of the nt type arguments to the property, this contains a mapping ets_i ($1 \leq i \leq nt$) from the free variables in T to Coq datatypes so that we can define the decoding $\lfloor T\rfloor_i$ of T. As explained in Section 2.5, each function argument $\texttt{pfn}\langle T\rangle_j$ ($1 \leq j \leq nx$) requires a similar environment with a mapping for each of its np type arguments; this environment is given by $\underset{\sim\sim}{(ets)}_j$.

efs As explained in Section 2.6, each function argument $\texttt{pfn}\langle T\rangle_j$ requires an environment *ef* containing functions for the free variables in T; *efs* is a tuple of nx such environments, one for each argument $\texttt{pfn}\langle T\rangle_j$.

```
(* Environment containing proofs for all free variables *)
Definition envp (nt nx np nv : nat)
  (ek : envk nv) (Pt : PolyType np)
  (Pp : PolyProp nt nx Pt) (ets : envts nt nv ek)
  (fns_i : ∀ i, gtupleTS (kit (getS i ek) Pt)
   (reindex_tuple (idxs Pp) (replace_fvs (tvar nv ek i) ets))) :=
 gtupleS (fun i => specProp' (tvar nv ek i) Pp ets (fns_i i))
        (elements_of_index nv).

(* Proof specialization for open types *)
Lemma specProof' (nt nx np nv : nat) (k : kind) (ek : envk nv)
  (t : type nv ek k) (Pt : PolyType np) (pfn : PolyFn Pt)
  (Pp : PolyProp nt nx Pt) (prf : PolyProof pfn Pp)
  (ets : envts nt nv ek)
  (efs : gtupleTS (fun x' => envf nv ek Pt x')
    (reindex_tuple (idxs Pp) ets))
  (ep : envp Pp (fun i => cst (tvar nv ek i) pfn (idxs Pp) ets efs))
  : specProp' t Pp ets (cst t pfn (idxs Pp) ets efs).
Proof.
  (* See Coq sources.
     The individual cases are explained in the text. *)
Defined.

(* Proof specialization for closed types *)
Definition specProof (nt nx np : nat) (k : kind) (t : closed_type k)
  (Pt : PolyType np) (pfn : PolyFn Pt)
  (Pp : PolyProp nt nx Pt) (prf : PolyProof pfn Pp)
  : specProp t Pp (cst_closed t pfn (idxs Pp)) :=
 specProof' t prf (ets_tt nt)
  (create_empty_gtup (envts np 0 tt) nx
    (reindex_tuple (idxs Pp) (ets_tt nt))) tt.
```

Figure 3.2: Proof Specialization

ep Finally, the definition of proof specialization assumes the existence of a proof p_A for each free variable A. In the formalization, environment *ep* contains a proof that the property holds at type A for each free variable A in T.

Figure 3.2 shows the formal statement (`specProof'`) that given an open type t, a polytypic proof *prf* over a polytypic function *pfn*, and given the environments *ets*, *efs* and *ep*, we can specialize the proof to t. The proof is by induction on t, as expected. We do not show the full Coq proof here (it can be found in the sources). Instead, we will discuss the individual cases of the proof below.

Since users will mostly be interested in proofs over closed types, we also provide a lemma (`specProof`) which states that for a closed type t and a polytypic proof *prf* over a polytypic function *pfn*, we can specialize the proof to t; `specProof` simply calls `specProof'` with the appropriately constructed empty environments.

3.5.1 Constants

The case for constants is given by the user except that, as explained in Section 3.3, we need a weakening lemma that says

$$\begin{aligned} &\mathtt{Pp}\langle k\rangle\ (\lfloor\varnothing \vdash \mathtt{C} : k\rfloor_0,\ldots)\ (\mathtt{pfn}\langle\varnothing \vdash \mathtt{C} : k\rangle,\ldots) \\ =\ &\mathtt{Pp}\langle k\rangle\ (\lfloor\Gamma \vdash \mathtt{C} : k\rfloor_0,\ldots)\ (\mathtt{pfn}\langle\Gamma \vdash \mathtt{C} : k\rangle,\ldots). \end{aligned}$$

The proof of this lemma was also given in Section 3.3.

3.5.2 Variables

Recall from Section 2.3 that variables in our universe are represented by De Bruijn indices. To construct the proof for a free variable i, we simply look up the ith element in environment *ep*. As for term specialization (Section 2.6), the trickiest part is to define the type of *ep*. Informally, the ith element in *ep*, corresponding to the proof for the ith variable, has type

$$\mathtt{Pp}\langle k\rangle\ (\lfloor i\rfloor_1,\ldots,\lfloor i\rfloor_{nt})\ (\mathtt{pfn}\langle i\rangle_1,\ldots,\mathtt{pfn}\langle i\rangle_{nx})$$

The formal definition of *ep* is given in Figure 3.2. The *construction* of *ep* will be considered when we discuss lambda abstraction in Section 3.5.4.

3.5.3 Application

For the specialization of a proof `prf` of type `Pp` to an application $(T\ U)$, we two induction hypothesis for the types T and U:

$$\begin{aligned} &\mathtt{IH}_T : \forall(A_1,\ldots,A_{nt})\ (g_1,\ldots,g_{nx})\ . \\ &\quad \mathtt{Pp}\langle k_1\rangle\ (A_1,\ldots,A_{nt})\ (g_1,\ldots,g_{nx}) \to \\ &\quad \mathtt{Pp}\langle k_2\rangle\ (\lfloor T\rfloor_1\ A_1,\ldots,\lfloor T\rfloor_{nt}\ A_{nt}) \\ &\qquad (\mathtt{app\_fs}\ (\mathtt{pfn}\langle T\rangle_1,\ldots,\mathtt{pfn}\langle T\rangle_{nx})\ (g_1,\ldots,g_{nx})) \\ &\mathtt{IH}_U : \mathtt{Pp}\langle k_1\rangle\ (\lfloor U\rfloor_1,\ldots,\lfloor U\rfloor_{nt})\ (\mathtt{pfn}\langle U\rangle_1,\ldots,\mathtt{pfn}\langle U\rangle_{nx}) \end{aligned}$$

and we need to prove:

$$\mathtt{Pp}\langle k_2\rangle\ (\lfloor T\ U\rfloor_1,\ldots,\lfloor T\ U\rfloor_{nt})\ (\mathtt{pfn}\langle T\ U\rangle_1,\ldots,\mathtt{pfn}\langle T\ U\rangle_{nx})$$

If we instantiate $(A_1,\ldots,A_{nt})$ by $(\lfloor U\rfloor_1,\ldots,\lfloor U\rfloor_{nt})$ and
$(g_1,\ldots,g_{nx})$ by $(\mathtt{pfn}\langle U\rangle_1,\ldots,\mathtt{pfn}\langle U\rangle_{nx})$ in $\mathtt{IH}_T$ and then apply this to $\mathtt{IH}_U$ we get

a term of type

$$\begin{array}{l}\texttt{Pp}\langle k_2\rangle\ (\lfloor T\rfloor_1\ \lfloor U\rfloor_1,\ldots,\lfloor T\rfloor_{nt}\ \lfloor U\rfloor_{nt})\\ \quad(\texttt{app\_fs}\ (\texttt{pfn}\langle T\rangle_1,\ldots,\texttt{pfn}\langle T\rangle_{nx})\ (\texttt{pfn}\langle U\rangle_1,\ldots,\texttt{pfn}\langle U\rangle_{nx}))\end{array}$$

To get the type we actually need we specify two conversion lemmas. The first conversion is fairly straight-forward, and its proof can be found in Section 2.6.3.

Lemma 10

$$\forall\ T\ U, (\lfloor T\rfloor_1\ \lfloor U\rfloor_1,\ldots,\lfloor T\rfloor_{nt}\ \lfloor U\rfloor_{nt}) = (\lfloor T\ U\rfloor_1,\ldots,\lfloor T\ U\rfloor_{nt})$$

This is the essence of Lemma 2 which we have proved in Section 2.6.3. The second lemma is a little trickier:

Lemma 11 (`convert_tapp_specProof`)

$$\begin{array}{l}\forall\ T\ U,\\ (\texttt{app\_fs}\ (\texttt{pfn}\langle T\rangle_1,\ldots,\texttt{pfn}\langle T\rangle_{nx})\ (\lfloor U\rfloor_1,\ldots,\lfloor U\rfloor_{nt}))\\ \simeq_{(\texttt{Pt}\langle k_2\rangle\ \underset{1}{\utilde{(\lfloor T\rfloor_1\ \lfloor U\rfloor_1,\ldots)}}\times\cdots),(\texttt{Pt}\langle k_1\rangle\ \underset{1}{\utilde{(\lfloor T\ U\rfloor_1,\ldots)}}\times\cdots)}\\ (\texttt{pfn}\langle T\ U\rangle_1,\ldots,\texttt{pfn}\langle T\ U\rangle_{nx})\end{array}$$

The proof of this lemma involves some manipulation of heterogeneous equalities. Note that Lemma 10, in addition to proving the first argument tuples equal, also proves that the two types involved in the heterogeneous equality in Lemma 11 are equal.

3.5.4 Lambda Abstraction

For the specialization of a lambda abstraction $\Lambda A\ .\ T$ we get the induction hypothesis for the body of the abstraction:

$$\texttt{IH}_T : \texttt{Pp}\langle k_2\rangle\ (\lfloor T\rfloor_1,\ldots,\lfloor T\rfloor_{nt})\ (\texttt{pfn}\langle T\rangle_1,\ldots,\texttt{pfn}\langle T\rangle_{nx})$$

for suitably extended environments *ets*, *efs* and *ep* (not shown in the informal notation). We need to prove:

$$\texttt{Pp}\langle k_1\to k_2\rangle\ (\lfloor\Lambda A\ .\ T\rfloor_1,\ldots,\lfloor\Lambda A\ .\ T\rfloor_{nt})\ (\texttt{pfn}\langle\Lambda A\ .\ T\rangle_1,\ldots,\texttt{pfn}\langle\Lambda A\ .\ T\rangle_{nx})$$

We know that the specialization $\texttt{Pp}\langle k_1 \rightarrow k_2\rangle$ takes the form

$$\begin{array}{l}
\forall A_1 \ldots A_{nt}\ (g_1, \ldots, g_{nx})\ . \\
\quad \texttt{Pp}\langle k_1\rangle\ (A_1, \ldots, A_{nt})\ (g_1, \ldots, g_{nx}) \rightarrow \\
\quad \texttt{Pp}\langle k_2\rangle\ (\lfloor \Lambda A\ .\ T\rfloor_1\ A_1, \ldots, \lfloor \Lambda A\ .\ T\rfloor_{nt}\ A_{nt}) \\
\quad\ (\texttt{app\_fs}\ (\texttt{pfn}\langle \Lambda A\ .\ T\rangle_1, \ldots, \texttt{pfn}\langle \Lambda A\ .\ T\rangle_{nx})\ (g_1, \ldots, g_{nx}))
\end{array}$$

Recall that for each free variable A in T, we need

- A set of nt types, given by *ets*, which is used to define the decoding of T, $\lfloor T\rfloor_i$ $(1 \leq i \leq nt)$
- For each of the nx function arguments to the property, a function that handles occurrences of terms of type A, given by *efs*
- A proof of the property at A, given by *ep*

In the body of the abstraction, we have one additional free variable, so we will need to extend these three environments: we add $(A_1, \ldots, A_{nt})$ to *ets*, $(g_1, \ldots, g_{nx})$ to *efs* and the proof of the property $\texttt{Pp}\langle k_1\rangle\ (A_1, \ldots, A_{nt})\ (g_1, \ldots, g_{nx})$ to *ep*.

Unfortunately, extending these environments is not quite as trivial as it may seem. The original environment *ep* contains proofs of type

$$\texttt{Pp}\langle k\rangle\ (\lfloor i\rfloor_1, \ldots, \lfloor i\rfloor_{nt})\ (\texttt{pfn}\langle i\rangle_1, \ldots, \texttt{pfn}\langle i\rangle_{nx})$$

for each type variable i of kind k, where the decoding is interpreted with respect to the original environment *ets*. However, in the body of the lambda abstraction each of these variables is shifted and is now known as $i+1$; variable 0 refers to the variable bound by the lambda. That means that we need to convert every proof in the original *ep* environment to a proof of type

$$\texttt{Pp}\langle k\rangle\ (\lfloor i+1\rfloor_1, \ldots, \lfloor i+1\rfloor_{nt})\ (\texttt{pfn}\langle i+1\rangle_1, \ldots, \texttt{pfn}\langle i+1\rangle_{nx})$$

where the decoding is now interpreted with respect to the extended environment *ets*. This involves proving that[5]

Lemma 12 *For each variable i of kind k*

$$\texttt{pfn}\langle i\rangle_1 \simeq_{\texttt{Pt}\langle k\rangle \utilde{(\lfloor i\rfloor_1, \ldots, \lfloor i\rfloor_{nt})}_1, \texttt{Pt}\langle k\rangle \utilde{(\lfloor i+1\rfloor_1, \ldots, \lfloor i+1\rfloor_{nt})}_1} \texttt{pfn}\langle i+1\rangle_1$$

where the left side of the equality is interpreted with respect to the original environments ets and efs, and the right side is interpreted with respect to the extended environments.

[5]In the abstraction case for term specialization, we have a similar but simpler problem, where we needed to prove only that the two types in this heterogeneous equality are equal.

Though this lemma may look innocent, it is in fact the most difficult proof in the entire formalization, and we needed to modify term specialization slightly to make the proof feasible. The difficulty comes from the many calls to `convert` that are generated by term specialization, so that the proof involves a lot of reasoning about various heterogeneous equalities. By making sure that these calls to convert are applied at a smaller granularity, the reasoning in proof specialization is somewhat simplified. A slightly different choice of universe might make it possible to reduce the number of places where we need conversion lemmas; we will come back to this in the section on related work.

Once all environments have been extended we need to apply the induction hypothesis $\mathtt{IH}_T$, but first we will need two conversion lemmas to get a proof of the correct type. The first lemma is part of the proof of Lemma 3 in Section 2.6.4:

Lemma 13

$$\forall\ A_1 \ldots A_{nt}\ T, (\lfloor \Lambda A\ .\ T \rfloor_1\ A_1, \ldots, \lfloor \Lambda A\ .\ T \rfloor_{nt}\ A_{nt}) = (\lfloor T \rfloor_1, \ldots, \lfloor T \rfloor_{nt})$$

where each $\lfloor T \rfloor_i$ *is decoded with ets extended as described above.*

The second conversion lemma we need deals with the function arguments:

Lemma 14 (`convert_tlam_specProof`)

$$\mathtt{app\_fs}\ (\mathtt{pfn}\langle \Lambda A\ .\ T\rangle_1, \ldots, \mathtt{pfn}\langle \Lambda A\ .\ T\rangle_{nx})\ (\mathtt{pfn}\langle A\rangle_1, \ldots, \mathtt{pfn}\langle A\rangle_{nx})$$
$$\simeq_{(\mathtt{Pt}\langle k_1 \to k_2\rangle\ \underbrace{(\lfloor \Lambda A\ .\ T \rfloor_1\ A_1, \ldots)}_{1} \times \cdots), (\mathtt{Pt}\langle k_1 \to k_2\rangle\ \underbrace{(\lfloor T \rfloor_1, \ldots)}_{1} \times \cdots)}$$
$$(\mathtt{pfn}\langle T\rangle_1, \ldots, \mathtt{pfn}\langle T\rangle_{nx})$$

Proof. Again, this proof is mostly a matter of juggling with heterogeneous equalities. □

3.6 Examples of Polytypic Properties and Proofs

In this section we will show a few more polytypic properties and their proofs.

3.6.1 Fusion Law for `count`

First, we have the fusion law for `count`:

```
(* h is a morphism on natural numbers *)
Definition morph (h : nat → nat) :=
 h 0 = 0 /\ ∀ i j, h (i + j) = h i + h j.

Definition Fusion : PolyProp 1 2 Count :=
 polyProp 1 2 Count
```

```
    ((1), (1))
    (fun T (f1, f2) =>
      ∀ h : nat → nat, morph h → ∀ x : T, h (f1 x) = f2 x).
```

We can specialize this property, as explained in Section 3.4, to the type `fork` : ⋆ → ⋆ and apply it to two instances of `count`:

```
Eval compute in
 specProp fork Fusion (count⟨fork⟩, count⟨fork⟩).
= ∀ (A : Set) (f g : A → nat)
   (Hx : ∀ h : nat → nat, morph h → ∀ x : A, h (f x) = g x),
   ∀ h : nat → nat, morph h → ∀ p : A × A,
    h (count⟨fork⟩ A f p) = count⟨fork⟩ A g p
```

This might look a little obscure, but if we take g to be $h \circ f$ (as is implied by the condition `Hx` above), the result will look more like the familiar fusion law for count:

```
= ∀ (A : Set) (f : A → nat) (h : nat → nat),
    morph h → ∀ p : A × A,
     h (count⟨fork⟩ A f p) = count⟨fork⟩ A (h . f) p
```

Note that this fusion law for `count` is rather limited, since the only functions that qualify as morphism are functions that multiply their argument by a given constant c.

To prove that this fusion law holds for count, we must provide the proofs for each of the type constants. This proof, again, follows nearly the same structure as the polytypic proofs `map_Id` and `map_Comp`.

```
Lemma count_Fusion : PolyProof count Fusion.
Proof.
(* This is a polyProof *)
  apply (polyProof count Fusion);
(* Solve case for unit and int by auto tactic *)
  compute -[plus]; auto; intros.
(* Case for products *)
  destruct x;
   rewrite <- (H h H1 H2); rewrite <- (H0 h H1 H2); auto.
(* Case for sums *)
  destruct x;
   [rewrite <- (H h H1 H2) | rewrite <- (H0 h H1 H2)] ; auto.
Defined.
```

3.6.2 Commutativity of Equality

As a second example, consider the property that equality is commutative:

```
Definition Commutative : PolyProp 1 1 Compare :=
 polyProp 1 1 Compare
  ((1))
  (fun T f => ∀ x y : T, f x y = f y x).
```

As we have said before, this is a more general property than the one we want to prove: it holds for any f of the correct type, whereas we want to prove the property for f = `equal`$\langle T \rangle$:

```
∀ x y : T, equal⟨T⟩ x y = equal⟨T⟩ y x
```

Specializing this property to `fork` and applying it to an instance of `equal` gives us exactly what we would expect: provided that the f on the the elements of the pair is commutative, the equality function for the pair is also commutative.

```
Eval compute in specProp fork Commutative equal⟨fork⟩.
= ∀ (A : Set) (f : A → A → bool)
    (Hxy : ∀ x y : A, f x y = f y x),
   ∀ x y : A × A, equal⟨fork⟩ A f x y = equal⟨fork⟩ A f y x
```

The polytypic proof that equality is commutative is slightly different from the proofs we have seen so far. In the case for integers, we need to prove that integer equality (`Zeq_bool`) is commutative. Unfortunately this lemma is not part of the Coq library on integers, so we need to provide this proof ourselves. In the case for products and sums, we are comparing two pairs/sums x and y, so we need to destruct both of these; whereas in other proofs we only had a single pair x to deal with.

```
Definition eq_Comm : PolyProof equal Commutative.
Proof.
  apply (polyProof equal Commutative);
   compute -[Zeq_bool]; auto; intros.
(* tint *)
  unfold Zeq_bool.
  puts (Zcompare_antisym x y); rewrite <- H.
  unfold CompOpp; elim (x ?= y)%Z; auto.
(* tprod *)
  destruct x; destruct y; rewrite <- H; rewrite H0; auto.
(* tsum *)
  destruct x; destruct y; auto.
Defined.
```

3.6.3 Non-Commutativity of `less_than`

Even though `less_than` and `equal` have the same type, they do have different properties. In the previous section we have shown that equality is commutative, but the opposite property holds for `less_than`: if $x < y$ then not $y < x$. This property can be defined as:

```
Definition NotCommutative : PolyProp 1 1 Compare :=
  polyProp 1 1 Compare
    ((0))
    (fun T f => ∀ x y : t, f x y = true → f y x = false).
```

To prove that this property holds for `less_than` we must instantiate the function f by `less_than`$\langle T\rangle$, and then prove the property for each of the type constants:

```
Definition lt_NotComm : PolyProof less_than NotCommutative.
Proof.
 apply (polyProof less_than NotCommutative);
  compute -[Zlt_bool andb]; auto; intros.
(* tint *)
 case_eq (Zlt_bool y x); auto; intro.
 rewrite <- Zlt_is_lt_bool in *.
 apply False_ind; omega.
(* tprod *)
 destruct x; destruct y.
 rewrite andb_true_iff in H1; destruct H1.
 rewrite H; auto.
(* tsum *)
 destruct x; destruct y; auto.
Defined.
```

This proof is slightly more involved than our previous examples, mainly because the occurrence of the &&-operator (`andb`) in the definition of `less_than`, which requires a few more unfolding steps.

3.6.4 `zero` is Idempotent

As a last example we will consider the property and proof that `zero` is idempotent, i.e.

$$\texttt{zero}\,(\texttt{zero}\,x) = \texttt{zero}\,x$$

In Coq, we can define this property as follows:

```
Definition Idempotent : PolyProp 1 3 Zero :=
 polyProp 1 3 Zero
  ((0), (0), (0))
  (fun T f1 f2 f3 => ∀ x : T, f1 (f2 x) = f3 x).
```

We can prove that this property holds when f_1, f_2 and f_3 are all instantiated to `zero`$\langle T\rangle$ by providing the proofs for each of the type constants. This proof is very straight-forward and follows the same structure as our earlier examples:

```
Definition zero_Idempotent : PolyProof zero Idempotent.
Proof.
  apply (polyProof zero Idempotent);
   compute; intros; auto.
(* tprod *)
  destruct x; rewrite H; rewrite H0; reflexivity.
(* tsum *)
  destruct x; [rewrite H | rewrite H0]; auto.
Defined.
```

Chapter 4

Recursion

4.1 Introduction

The careful reader will have noticed that our universe does not contain any notion of recursion. By far the easiest way to add recursive datatypes would be to add recursion directly to our type universe, as is done in (Hinze, 2000b). However, since Coq does not support general recursion at the type level, we would not be able to define a type decoder for such a universe. Restricting our universe to strictly positive types might provide a way around this problem, as shown in (Morris et al., 2006). Unfortunately, this approach is limited to first-order kinds and there is no obvious way to extend it to support types of higher-order kinds. Since our aim is to be able to reason about Generic Haskell-style programs, which include higher-order types, we will not use this solution. We discuss the ideas in (Morris et al., 2006) in more detail in Section 5.4.2.

Generic Haskell lives in the category of complete partial orders and strict continuous functions. In this category initial algebras and final coalgebras—and therefore inductive and coinductive datatypes—coincide (Fokkinga and Meijer, 1991), meaning that the `List` datatype in Haskell captures both finite and infinite lists. In order to more closely model Generic Haskell datatypes in Coq we will use coinduction to define (co)recursive datatypes.

The universe in Generic Haskell or Generic Clean does not include a general recursion operator either (Löh, 2004, Section 7.5.1), (Alimarine, 2005, Chapter 2). Instead, recursion happens at the term level during the translation to and from the structural representation of the datatype. We would like to use a similar solution to recursion in our system. As an example, let us have a look at the coinductive type `list`:

```
CoInductive list (A : Set) : Set :=
 | nil : list A
 | cons : A → list A → list A.

Definition list_kind : kind := karr star star.

Definition list_struct : type 1 (list_kind, tt) list_kind :=
  let var := tvar 2 (star, (list_kind, tt))
  in tlam (1 + var None × var (Some None) @ var None).

Definition list' (A : Set) : Set :=
  decT list_struct (list, tt) A.
```

Although the notation would benefit from some syntactic sugar, hopefully it is clear that `list_struct` corresponds to the structural type $\lambda A\ .\ 1 + A \times B\ A$, where B is a free variable of kind $\star \to \star$. We can decode this type to an actual Coq type using our type decoder, where we pass the coinductive `list` type as the decoding of the free variable B. In other words, the decoded type `list'` corresponds to the type $\lambda A\ .\ 1 + A \times \texttt{list}\ A$.

The two types `list` and `list'` are isomorphic, and this isomorphism is witnessed by an embedding-projection pair:

```
Definition fromList (A : Set) (l : list A) : list' A :=
  match l with
  | nil => inl _ tt
  | cons a l' => inr _ (a, l')
  end.

Definition toList (A : Set) (l : list' A) : list A :=
  match l with
  | inl _ => nil
  | inr (a, l') => cons a l'
  end.
```

where we can prove that $\texttt{fromList} \circ \texttt{toList} = \texttt{id} = \texttt{toList} \circ \texttt{fromList}$.

Given this structural representation for lists, we can apply our existing definition of term specialization to get the polytypic `map` function specialized to the `list_struct` type:

```
Definition mapList' :=
  specTerm' list_struct map ((list, tt), ((list, tt), tt)).
```

Coq is now able to tell us that the type of `mapList'` is[1]

```
(∀ A B : Set, (A → B) → list A → list B) →
 ∀ A B : Set, (A → B) → list' A → list' B
```

We can use this definition of `mapList'` to coinductively define `map` specialized to `list`

```
CoFixpoint mapList (A B : Set) (f : A → B) (l : list A)
   : list B :=
  toList (mapList' mapList f (fromList l)).
```

[1]`mapList'` actually requires an environment containing the map function on `list` as its first argument. To aid readability, we will assume here that its first argument is simply this function, not wrapped in any environment.

This is exactly the sort of definition that would be used in Generic Haskell or Generic Clean, but unfortunately the definition is rejected by Coq.

As we have seen, recursive functions (on inductive datatypes) must terminate, and Coq guarantees this by posing a syntactic restriction on the way functions may be defined: they have to be "guarded-by-destructors". Intuitively, this constraint imposes a bound on the number of possible recursive calls using the size of the term given as input. For functions that produce terms in coinductive types restrictions are instead placed on the way the output data is produced. A corecursive call is accepted only if some information has been produced in the result in the form of a constructor. The terminology is that calls must be "guarded-by-constructors" (Bertot, 2005). We can describe guardedness in two steps (Bertot and Komendantskaya, 2008):

1. A position is *pre-guarded* if it occurs as the root of the function body, or if it is a direct subterm of a pattern-matching construct or a conditional statement, which is itself in a pre-guarded position.

2. A position is *guarded* if it occurs as a direct subterm of a constructor for the coinductive type that is being defined and if this constructor occurs in a pre-guarded or guarded position.

A corecursive function is guarded if all corecursive calls occur in guarded positions.

Unfortunately, the definition of `mapList` above is not (obviously) guarded since `mapList` appears in a non-guarded position, and Coq rejects its definition.

In this chapter we will discuss various ways to ensure coinductive functions such as `mapList` pass the scrutiny of the Coq guardedness checker, eventually settling on a minor modification to the guardedness checker in Section 4.3.2. Section 4.6 then goes on to show a slightly more complicated example where the function is partial and where the guardedness checker is not so easily satisfied.

4.2 Partiality

One way to define `mapList` coinductively is to make use of the partiality monad (Capretta, 2005). The partiality monad allows us to delay computation:

```
CoInductive Delay (A : Set) : Set :=
 | Now : A → Delay A
 | Later : Delay A → Delay A.
```

This monad can be thought of as capturing the essence of productivity: the productivity requirement for a function can be satisfied simply by guarding each corecursive call by the `Later` constructor. Since this is a monad we should define the return and bind operators; `returnD` returns the result of a computation (wrapped in the monad) and `bindD` strings two computations together:

```
Definition returnD (A : Set) (a : A) := Now a.

CoFixpoint bindD (A B : Set) (d : Delay A)
  (k : A → Delay B) : Delay B :=
 match d with
 | Now a => k a
 | Later d' => Later (bindD d' k)
 end.
```

We can define `bottom` (the undefined value) by delaying the computation indefinitely:

```
CoFixpoint bottom (A : Set) : Delay A := Later (bottom A).
```

The exciting feature of the partiality monad is that it allows us to define a general fixpoint combinator for `mapList`, whose type is given by:

```
Definition lfp_mapList
  (h : (∀ C D, (C → Delay D) → list C → Delay (list D)) →
        ∀ C D, (C → Delay D) → list C → Delay (list D))
  : ∀ A B, (A → Delay B) → list A → Delay (list B)
```

Given this operator we can give a straight-forward definition of `mapList`, using the definition of `mapList'` from Section 4.1:

```
Definition mapList (A B : Set) (f : A → Delay B)
  (l : list A) : Delay (list B) :=
 lfp_mapList (fun rec A' B' f' l' =>
  bindD (mapList' (rec, tt) f' (fromList l'))
        (fun result => returnD (toList result))) f l.
```

Unfortunately, the type of `lfp_mapList` and its definition depend on the type of the polytypic function we want to define, and the type we want to specialize it to. This means that we must define a separate fixpoint operator for each specialization of a polytypic function. However, the definition of the general fixpoint operator is fairly trivial, and it might be possible to define it polytypically.

Another slight disadvantage of the use of the partiality monad is that all polytypic functions must now be defined in monadic style. For example, the case for products in the polytypic `map` function will take the form:

```
fun (A B : Set) (f : A -> Delay B)
    (C D : Set) (g : C -> Delay D) (x : A × C) =>
 let (a, c) := x in
  bindD (f a) (fun b => (
  bindD (g c) (fun d => (
  returnD (b, d)))))
```

A more important disadvantage is that this approach can not be extended to deal with properties and proofs. While it is possible to define a function that delays the construction of a list, it is *not* possible to delay the construction of a *proof*. If we were able to do that, we could prove any property P by delaying the proof indefinitely: bottom P. Moreover, proofs over functions in the partiality monad are far from trivial.

4.3 Guardedness

In the previous section we have shown that we can give a guarded coinductive definition of mapList by wrapping it in a general fixpoint operator, which we can construct using the partiality monad. In this section we will take a closer look at our original definition of mapList:

```
CoFixpoint mapList (A B : Set) (f : A → B) (l : list A)
  : list B :=
 toList (mapList′ mapList f (fromList l)).
```

The reason that it does not pass Coq's guardedness check is because Coq does not sufficiently unroll the definition to be able to see that it is guarded. In this section we propose two ways to resolve this to get a valid definition of mapList, but first we will show that the definition is in fact guarded. Let us first unroll the definition of mapList′:

```
mapList′ =
 fun (rec : ∀ A B : Set, (A → B) → list A → list B)
     (A B : Set) (f : A → B) (l : list′ a) =>
 match l with
 | inl u => inl _ u
 | inr (a, l′) => inr _ (f a, rec f l′)
 end
```

The recursive call to mapList—which corresponds to the argument rec in the definition of mapList′—is not obviously guarded by a constructor of list. However, guardedness is checked with respect to various reductions, and Coq's guardedness checker reduces the definition of mapList to:

```
match
 match
  match l with
   | nil => inl _ tt
   | cons a l′ => inr _ (a, l′)
  end
 with
 | inl _ => inl _ tt
 | inr (a, l′) => inr _ (f a, mapList f l′)
 end
with
| inl _ => nil
| inr (a, l′) => cons a l′
end
```

Looking at this more closely, we see that the corecursive call to `mapList` only occurs when the input takes the shape `cons a l'`. The call to `fromList` (the innermost match) will convert this to

```
inr _ (a, l')
```

This is the input to `mapList'` (the second match), where the recursive call takes place on the tail of the list to give us

```
inr _ (f a, mapList f l').
```

Finally, this will pass through `toList` (the outermost match) and be converted to

```
cons (f a) (mapList f l')
```

Considering that this is the only occurrence of the corecursive call, we know it will always be guarded by the `cons` constructor. Unfortunately, Coq's guardedness checker is not quite clever enough to detect this, and the definition is rejected.

4.3.1 CPS Transforms

One way to convince Coq that the above coinductive definition of `mapList` is guarded is to use CPS transforms (Plotkin, 1975)—a function is in continuation-passing style (CPS) if instead of returning a value it takes an explicit continuation function which is applied to the result of the function.[2]

We must modify `mapList'` slightly so that it does not return the resulting `list' B` directly, but instead applies a continuation `K`. We can then move the application of `K` into the branches of the match construct, ensuring that Coq will unfold the application of `K` (since Coq never unfolds an application when the function is applied to an argument of the form `match ... end`, we need to move the application inside the branches).

```
Definition CPS_mapList'
  (rec : ∀ A B : Set, (A → B) → list A → list B)
  (A B R : Set) (f : A → B) (l : list' A)
  (K : list' B → R) : R :=
 match l with
 | inl u => K (inl _ u)
 | inr (a, l') => K (inr _ (f a, rec f l'))
 end.
```

We can use `CPS_mapList'` to define `CPS_mapList`, much as we did before, except that we now pass `toList` as the continuation:

```
CoFixpoint CPS_mapList (A B : Set) (f : A → B)
  (l : list A) : list B :=
 CPS_mapList' CPS_mapList f (fromList l) (@toList B).
```

[2]Thanks to Bruno Barras and Russell O'Connor on the Coq mailing list for suggesting this possibility.

Because the application of `toList` is now moved inside the branches of the match construct associated with `CPS_mapList'`, Coq is able to reduce the definition of `CPS_mapList` to

```
CPS_mapList f l =
match
 match l with
 | nil => inl _ tt
 | cons a l' => inr _ (a, l')
 end
with
| inl _ => nil
| inr (a, l') => cons (f a) (CPS_mapList f l')
end
```

and this definition is accepted because the corecursive call is clearly guarded by the `cons` constructor.

Although this method shows that the corecursive call in `mapList` is indeed guarded as long as the term can be sufficiently unrolled by Coq, the integration of this method with our existing development is far from trivial. In particular, we would need to modify the return type of term specialization to incorporate the CPS transform.

As discussed in Chapter 2 the type of *term* specialization is given by *type* specialization:

```
specTerm (t : closed_type k) (pfn : PolyFn Pt)
 : specType t Pt
```

and type specialization for the polytypic type `Map` takes the type for kind $\star$ as given by the user:

```
Definition Map : PolyType 2 :=
 polyType 2 (fun A B : Set => A → B).
```

and specializes this to kind $\star \to \star$ to get the correct type for `map` acting on (the structural representation of) lists. To incorporate the CPS transform, we need to find a new type `Map`, which gives us the correct specialized type for `list'`. Again, just as for the use of the partiality monad in Section 4.2, the definition of polytypic types and functions given by the user must change. For partiality we needed to make all of these definitions monadic, here we need to incorporate the CPS transform in each of them. This is obviously not ideal. Furthermore, it is not obvious what the new polytypic type `Map` for kind $\star$ should be. We might consider the following definition:

```
Definition Map : PolyType 3 :=
 polyType 3 (fun A B R : Set => A → (B → R) → R).
```

However, its specialization to the type `list'` would be:

```
∀ A B R : Set, (A → (B → R) → R) →
 list' A → (list' B → list' R) → list' R
```

which is not at all what we want. Unfortunately, there does not seem to be a definition of `Map` that specializes to the correct type for `CPS_mapList'`, so we cannot easily modify type and term specialization to integrate CPS transforms.

4.3.2 μ-Reduction

Coq checks guardedness with regards to the input modulo the following reductions:

β-reduction (function application) Reduce an abstraction applied to a term to the body of the abstraction with its argument replaced by the term it is applied to: `(fun x ⇒ t) u` $\rightarrow_\beta$ `t[x/u]`.

δ-reduction (variable substitution) Substitute a variable or constant by its value in its context: `x` $\rightarrow_\delta$ `u` (where the environment contains `x := u`).

ι-reduction (destructor constructor) This reduction relation consists of two separate reductions. First, replace a pattern match applied to a constructor term by the appropriate branch; for example

```
match cons x xs with          →ι      P (cons x xs)
| nil => P nil
| cons y ys => P (cons y ys)
end
```

Second, unroll a function which is (structurally) recursive in its nth argument once, if a constructor argument is supplied for the nth argument:

```
fix plus (n m : nat) : nat :=
 match n with
 | O => m
 | S p => S (plus p m)
 end


→ι


fun n m : nat =>
 match n with
 | O => m
 | S p => S ((fix plus (n' m' : nat) : nat :=
              match n' with
              | O => m'
              | S p' => S (plus p' m')
              end) p m)
 end
```

Since `plus` is structurally recursive in its first argument, `plus (S n) m` $\rightarrow_{\iota,\delta,\beta}$ `S (plus n m)`, but `plus n (S m)` is not affected by ι-reduction.

ζ-reduction (local definitions) Given a local definition of a variable, substitute that variable by its value: **`let`** `x := u` **`in`** `t` $\rightarrow_\zeta$ `t[x/u]`.

In the previous section we have shown that the recursive call in the coinductive definition for `mapList` is in fact guarded, and that by applying a CPS transform to the result of `mapList'` we force Coq to reduce the definition sufficiently to satisfy the guardedness checker. In this section we will show how to modify the actual guardedness checker in the Coq source code in such a way that it reduces the definition of `mapList` sufficiently to pass the test.

It is important to note, however, that we do not modify the code that performs the guardedness check itself: the definition of guardedness does not change. We only apply an additional reduction before the function is checked for guardedness, which we will call μ-reduction.

Intuitively μ-reduction can be thought of as the collapsing of nested `match` statements, and is similar to the *case-of-case transformation* as implemented in the Glasgow Haskell Compiler (Peyton Jones, 1996, Section 5). For a more formal definition we will represent the `match` construct by

$$\text{match}_p\ e \text{ with } \langle f_1, f_2, \ldots \rangle$$

This is close to the internal representation of `match` in the Coq source code, modulo some syntactic differences to aid readability. The attribute p indicates the type of the match. For a scrutinee e of type t_1, p will return the type of the branches; that is, p takes the form $\lambda(x : t_1)\ .\ (t_2 : \texttt{Set})$. A match is *dependent* if the type of the branches depends on the value of the scrutinee—in other words, when x occurs free in t_2. If the match is not dependent (the type of each branch is the same) it is called a *simple* match.

The branches are represented by the functions $f_1, f_2, \ldots$, whose arguments correspond to the arguments of the constructor associated with that branch. For example, if we match on a term of type `list A`, the branch for the `nil` constructor will be a nullary function of type $(t_2\ \texttt{nil})$, and the branch for the `cons` constructor will be a binary function of the form $\lambda a\ l\ .\ e'$, where $e' : t_2\ a\ l$.

Figure 4.1 shows the definition of μ-reduction. It makes use of an operator $f \bullet_n g$ which corresponds to the composition of a function f with a function g of arity at least n. The definition assumes that g is a concrete function; i.e., of the shape $\lambda x_1\ .\ \lambda x_2\ .\ \cdots$. This is sufficient for our purposes. When applying this operator to a branch, n will be the arity of the constructor associated with that branch. In our examples we will leave n implicit for readability. To see how μ-reduction works, let us go back to our definition of `mapList` after β, δ, ι and ζ reductions have been applied:

```
match
 match
  match l with
  | nil => inl _ tt
```

μ-reduction is defined to be the smallest compatible closure of

$$\begin{aligned}&\text{match}_p\Big(\text{match}_{p'}\ e'\ \text{with}\ \langle f_1',\ldots\rangle\Big)\text{with}\ \langle f_1,\ldots\rangle\\ &\to_\mu\\ &\text{match}_{p\bullet p'}\ e'\text{with}\ \langle(\lambda x\ .\ \text{match}_p\ x\ \text{with}\ \langle f_1,\ldots\rangle)\bullet f_1',\ldots\rangle\end{aligned}$$

where

$$\begin{array}{llll} f & \bullet_0 & e & = f\ e \\ f & \bullet_{(n+1)} & (\lambda x\ .\ e) & = \lambda x\ .\ (f \bullet_n e) \end{array}$$

Figure 4.1: μ-Reduction

```
      | cons a l' => inr _ (a, l')
      end
    with
    | inl _ => inl _ tt
    | inr (a, l') => inr _ (f a, mapList f l')
    end
  with
  | inl _ => nil
  | inr (a, l') => cons a l'
  end
```

This definition does not pass the guardedness check. Internally, it can be represented by

$$\begin{array}{l}\text{match}_{\lambda\_:\,\text{list' }B\,.\,\text{list }B}\\ \quad\text{match}_{\lambda\_:\,\text{list' }A\,.\,\text{list' }B}\\ \quad\quad\text{match}_{\lambda\_:\,\text{list }A\,.\,\text{list' }A}\ l\\ \quad\quad\text{with}\ \langle\text{inl tt},\ \lambda a\ l'\ .\ \text{inr}\ (a,l')\rangle\\ \quad\text{with}\ \langle\lambda u\ .\ \text{inl tt},\ \lambda(a,l')\ .\ \text{inr}\ (f\ a,\text{mapList}\ f\ l')\rangle\\ \text{with}\ \langle\lambda u\ .\ \text{nil},\ \lambda(a,l')\ .\ \text{cons}\ a\ l'\rangle\end{array}$$

Figure 4.2 shows how this term can be simplified by applying mu-reduction twice, first to the inner-most two matches and then to the resulting outermost two matches.

Translating the result back to normal Coq notation gives us the definition of `mapList` that we would expect:

```
match l return list B with
| nil => nil
| cons a l' => cons (f a) (mapList f l')
end
```

When we modify Coq to check guardedness with respect to μ-reduction (in addition to the other reduction relations), Coq will be able to verify that the corecursive call in `mapList` is in fact guarded. See Appendix A for a *diff* file detailing our modifications

$$
\begin{array}{ll}
 & \text{match}_{\lambda\_ : \text{list' } B \,.\, \text{list } B} \\
 & \quad \text{match}_{\lambda\_ : \text{list' } A \,.\, \text{list' } B} \\
 & \quad\quad \text{match}_{\lambda\_ : \text{list } A \,.\, \text{list' } A}\ l \\
 & \quad\quad \text{with } \langle \text{inl tt},\ \lambda a\ l' \,.\, \text{inr}(a, l') \rangle \\
 & \quad \text{with } \langle \lambda u \,.\, \text{inl tt},\ \lambda(a, l') \,.\, \text{inr } (f\ a, \text{mapList } f\ l') \rangle \\
 & \text{with } \langle \lambda u \,.\, \text{nil},\ \lambda(a, l') \,.\, \text{cons } a\ l' \rangle \\
\rightarrow_{\mu} & \text{match}_{\lambda\_ : \text{list' } B \,.\, \text{list } B} \\
 & \quad \text{match}_{(\lambda\_ : \text{list' } A \,.\, \text{list' } B) \bullet (\lambda\_ : \text{list } A \,.\, \text{list' } A)}\ l \\
 & \quad \text{with } \langle (\lambda e \,.\, \text{match}_{\lambda\_ : \text{list' } A \,.\, \text{list' } B}\ e \text{ with } \langle \lambda u \,.\, \text{inl tt}, \ldots \rangle) \bullet (\text{inl tt}), \ldots \rangle \\
 & \text{with } \langle \lambda u \,.\, \text{nil},\ \lambda(a, l') \,.\, \text{cons } a\ l' \rangle \\
\rightarrow_{\beta,\iota} & \text{match}_{\lambda\_ : \text{list' } B \,.\, \text{list } B} \\
 & \quad \text{match}_{\lambda\_ : \text{list } A \,.\, \text{list' } B}\ l \\
 & \quad \text{with } \langle \text{inl tt},\ \lambda a\ l' \,.\, \text{inr } (f\ a, \text{mapList } f\ l') \rangle \\
 & \text{with } \langle \lambda u \,.\, \text{nil},\ \lambda(a, l') \,.\, \text{cons } a\ l' \rangle \\
\rightarrow_{\mu} & \text{match}_{(\lambda\_ : \text{list' } B \,.\, \text{list } B) \bullet (\lambda\_ : \text{list } A \,.\, \text{list' } B)}\ l \\
 & \text{with } \langle (\lambda e \,.\, \text{match}_{\lambda\_ : \text{list' } B \,.\, \text{list } B}\ e \text{ with } \langle \lambda u \,.\, \text{nil}, \ldots \rangle) \bullet (\text{inl tt}), \\
 & \quad (\lambda e \,.\, \text{match}_{\lambda\_ : \text{list' } B \,.\, \text{list } B}\ e \text{ with } \langle \ldots, \lambda(a, l') \,.\, \text{cons } a\ l' \rangle) \bullet \\
 & \quad\quad (\lambda a\ l' \,.\, \text{inr } (f\ a, \text{mapList } f\ l')) \rangle \\
\rightarrow_{\beta,\iota} & \text{match}_{\lambda\_ : \text{list } A \,.\, \text{list } B}\ l \\
 & \text{with } \langle \text{nil},\ \lambda a\ l' \,.\, \text{cons } (f\ a)\ (\text{mapList } f\ l') \rangle
\end{array}
$$

Figure 4.2: μ-Reduction Example

to the Coq source code.

We can now specialize the polytypic function `map` to the corecursive type `list` by specializing it to the non-recursive type `list'` and applying the appropriate functions from the embedding-projection pair.

4.4 Proofs

In Section 4.3.2, we have shown how coinduction can be used to define the specialization of a polytypic function to a recursive datatype. This section shows how to specialize proofs of polytypic properties in the same way.

Let us take a look at the proof that `map` preserves identities: `mapId`. First we need to modify our definition of this property to incorporate potentially infinite datatypes, like `list`. To prove that two lists are equal, we need to compare each corresponding pair of elements in the lists. If the lists are infinite, this cannot be done in finite time, and therefore the proof is non-terminating. Since non-terminating proofs are not allowed in Coq—we would be able to prove anything—we need a different notion of equality to compare potentially infinite lists: *bisimilarity*.

Whereas equality is an inductive relation, and therefore requires that any proof about equality terminates; bisimilarity on potentially infinite datatypes is a coinductive relation, requiring instead that all proofs are *productive*. In a proof involving a coinductive property, we must be able to show that we can always produce the next step in the proof. The bisimulation relations for `list` and `list'` are defined as follows:

```
CoInductive bisim_list (A : Set)
   : list A → list A → Prop :=
  | bisim_nil : bisim_list nil nil
  | bisim_cons : ∀ (a1 a2 : A) (l1 l2 : list A), a1 = a2 →
     bisim_list l1 l2 →
     bisim_list (cons a1 l1) (cons a2 l2).

Inductive bisim_list' (A : Set)
   : list' A → list' A → Prop :=
  | bisim_inl : ∀ u u' : unit,
     bisim_list' (inl _ u) (inl _ u')
  | bisim_inr : ∀ (a1 a2 : A) (l1 l2 : list A),
     a1 = a2 → bisim_list l1 l2 →
     bisim_list' (inr unit (a1, l1)) (inr unit (a2, l2)).
```

The bisimilarity relation for `list'` does not need to be coinductive, because its proofs will always terminate: we can provide both a proof that the heads of the lists are equal, and a proof that the tails are bisimilar (using `bisim_list`). Note that we use equality to compare the elements of the lists instead of bisimilarity, which restricts the lists that can be compared: we cannot show that two lists of lists are bisimilar. The reason is that we do not have a "generic" bisimilarity relation that we can use to compare elements of type A. We will discuss this problem in a little more detail in Section 6.3.4.

Using these bisimilarity relations, we can define `mapIdList'`, a proof that the function `mapList'` preserves identities, given that we have such a proof for the tail of the list:

```
Theorem mapIdList' (A : Set) (f : A → A) (l : list' A)
  (rec : ∀ (A : Set) (f : A → A) (l : list A),
   (∀ x : A, f x = x) → bisim_list (mapList f l) l)
  : (∀ x : A, f x = x) →
      bisim_list' (mapList' f l (@mapList A A)) l.
```

Ideally, this property and its proof will be provided by the specialization of the property `Id` and its proof `mapId` to `list'`. However, since unlike for equality we cannot give one definition of bisimilarity that can be used for any datatype, we cannot even *state* the polytypic property, much less give a polytypic proof for it. This can be solved by using type-indexed types, which is discussed in future work. In this section we will simply assume that `mapIdList'` is defined, and focus on the problem of (co)recursion only.

We can now prove the property `Id` specialized to `list` in a similar way as we have defined the function `mapList`:

```
Theorem mapIdList (A : Set) (f : A → A)
   (Hx : ∀ x : A, f x = x) (l : list A)
   : bisim_list (mapList f l) l.
Proof.
 cofix mapIdList; intros A f Hx l.
 rewrite (list_decomposition_lemma (mapList f l)); simpl.
 change (bisim_list (list_decompose
   (toList (mapList' (mapList, tt) f (fromList l)))) l).
 rewrite <- list_decomposition_lemma.
 rewrite <- to_from_id_list.
 apply (to_preserves_bisim_list
        (mapIdList' mapIdList _ Hx _)).
Defined.
```

Modulo some matching on equality proofs, this proof is equivalent to the corecursive function

```
CoFixpoint mapIdList (A : Set) (f : A → A)
  (Hx : ∀ x : A, f x = x) (l : list A)
   : bisim_list (mapList f l) l :=
 to_preserves_bisim_list
  (mapIdList' mapIdList f Hx (fromList l)).
```

and we can see that the proof follows the structure of `mapList`, the only difference is that the result is not guarded by `toList` but by `to_preserves_bisim_list`:

```
Definition to_preserves_bisim (A : Set) (l l' : list' A)
  (H : bisim_list' l l')
   : bisim_list (toList l) (toList l') :=
 match H in bisim_list' l l'
  return bisim_list (toList l) (toList l') with
 | bisim_inl _ _ => bisim_nil A
 | bisim_inr a b la lb Ha Hl => bisim_cons Ha Hl
 end.
```

The recursive call to `mapIdList` occurs as an argument to `mapIdList'`. Unrolling the definition of `to_preserves_bisim`, we see that the recursive call will match the argument `Hl` for the `bisim_inr` constructor. Since `Hl` only occurs guarded by `bisim_cons`, the above proof of `mapIdList` is guarded, provided that we use our definition of μ-reduction as described in Section 4.3.2.

4.5 Restrictiveness of Syntactic Guardedness

While the above proof of `mapIdList` is guarded, this guardedness very much depends on the actual proof given. A minor change to the proof can break guardedness. For example, if we change the proof slightly, using transitivity of the bisimilarity relation:

```
CoFixpoint bisim_list_trans (A : Set) (xs ys zs : list A)
  (pf : bisim_list xs ys)
   : bisim_list ys zs → bisim_list xs zs :=
Lemma mapIdList (A : Set) (f : A → A)
  (Hx : ∀ x : A, f x = x) (l : list A)
  : bisim_list (mapList f l) l.
Proof.
 cofix mapIdList ; intros A f Hx l.
 rewrite (list_decomposition_lemma (mapList f l)) ; simpl.
 change (bisim_list (list_decompose
  (toList (mapList' (mapList, tt) f (fromList l)))) l).
 rewrite <- list_decomposition_lemma.
 apply bisim_list_trans with (ys := toList (fromList l)).
 apply (to_preserves_bisim (mapIdList' _ _ mapIdList Hx)).
 rewrite to_from_id; apply bisim_id.
Defined.
```

This seems like an equally valid proof of mapIdList. However, this proof is *not* guarded. The reason for this is that although the recursive call is still guarded by the bisim_cons constructor as before, it is now also part of an argument to bisim_list_trans, which is a a corecursive function, and Coq does not check for guardedness of arguments to corecursive functions (see Section 2.2.5).

So we can indeed use coinduction to construct a proof of mapIdList, but we need to be very careful when constructing this proof. The syntactic guardedness check as implemented in Coq is very brittle, and will easily break. In particular, we have to make sure that the term containing the recursive call is not passed as an argument to a corecursive function or to another unknown function. This also means that any lemmas we use must be transparent instead of opaque, i.e. Coq must be able to access its proof to verify guardedness.

4.6 Count Example

In the previous section we have shown that the syntactic guardedness check used in Coq can be quite restrictive, and we need to be very careful to keep our definitions guarded. Related to that is the choice of codomain for our function, which we will deal with in this section.

As an example, consider the polytypic count function (see Section 2.7), whose type we gave as $T \to$ nat. As long as we are talking about induction and recursive (but finite) datatypes, this type is sufficient. However, as soon as we want to reason about coinductive types, we can no longer use nat as the codomain of the function. Counting the number of elements in an infinite list will result in an infinite number, which cannot be represented by nat.

We will need to make three modifications to the type of count: we need to make it coinductive, we need add an auxiliary datatype that includes a domain-specific constructor, and we need to add a step constructor. We will look at each of these modifications in turn.

4.6.1 Conatural Numbers

The first and most obvious modification that we need is to use conatural numbers rather than natural numbers: i.e., a datatype that describes both finite and infinite numbers. The conaturals can be defined as follows:

```
CoInductive coN : Set :=
 | coZ : coN
 | coS : coN → coN.
```

This allows us to represent infinite numbers, e.g.:

```
CoFixpoint inf := coS inf.
```

4.6.2 Domain-Specific Constructors

We can now reason about infinite numbers, but this is not sufficient. In Section 4.5 we have seen that syntactic guardedness poses some restrictions, and for some polytypic functions it is a little tricky to satisfy the guardedness checker. In particular, a corecursive call cannot be passed as an argument to a function. We run into this problem here when trying to specialize `count`. For example, when specializing `count` to trees, the non-leaves case gives us two corecursive calls to count each of the subtrees, the result of which we then want to add together:

```
add (count f t1) (count g t2)
```

However, this will make the corecursive calls to `count` arguments to the (corecursive) `add` function and we have seen in Section 4.5 that arguments to a corecursive function are never guarded. To solve this we need to define a domain-specific variation on conatural numbers that includes addition as a constructor:

```
CoInductive coN' : Set :=
 | coZ' : coN'
 | coS' : coN' → coN'
 | add' : coN' → coN' → coN'.
```

This will ensure that the corecursive calls to count are guarded by the `add'` constructor.

We need a flattening function that takes a `coN'` and flattens out all occurrences of `add'` to get a `coN`. In the next section we will show that this flattening function cannot be defined with the current definition of conatural numbers.

Whenever a polytypic function applies a coinductive function to its corecursive calls we will need to add a domain-specific datatype along with a flattening function to satisfy the guardedness checker.

4.6.3 Delaying Computation

To convert a domain-specific conatural number into a regular conatural number we need to define a function that flattens out all `add'` constructors. One way to define this function is as follows:

```
CoFixpoint flatten_coN's (ns : list coN') : coN :=
  match ns with
  | nil => coZ
  | n :: ns' =>
      match n with
      | coZ' => flatten_coN's ns'
      | coS' n' => coS (flatten_coN's (n' :: ns'))
      | add' m' n' => flatten_coN's (m' :: n' :: ns')
      end
  end.

Definition flatten_coN' (n : coN') : coN :=
  flatten_coN's (n :: nil).
```

However, two of the corecursive calls are not guarded in this definition (in the case for `coZ'` and `add'`), so this definition is rejected by Coq—and rightly so! Consider the case where the number we are trying to flatten consists only of add constructors: we would never produce a result. Such a number would result from trying to count the leaves in an infinitely branching tree without any leaves. Hence, this function is not productive. To make it productive, we can add a `tau` constructor to our codomain, which takes a role very similar to that of the `Later` constructor of the partiality monad discussed in Section 4.2: it allows us to delay computation, potentially indefinitely.

```
CoInductive coN : Set :=
 | coZ : coN
 | coS : coN → coN
 | tau : coN → coN.
```

We can then rewrite the flatten function so that it passes the Coq guardedness requirements:

```
CoFixpoint flatten_coN's (ns : list coN') : coN :=
  match ns with
  | nil => coZ
  | n :: ns' =>
      match n with
      | coZ' => tau (flatten_coN's ns')
      | coS' n' => coS (flatten_coN's (n' :: ns'))
      | add' m' n' => tau (flatten_coN's (m' :: n' :: ns'))
      end
  end.

Definition flatten_coN' (n : coN') : coN :=
  flatten_coN's (n :: nil).
```

4.6.4 Polytypic Count, Coinductively

We can now give a new definition of the polytypic count function that can deal with coinductive datatypes:

```
Definition co_Count : PolyType 1 :=
 polyType 1 (fun A => A → coN').

Definition co_count : PolyFn co_Count :=
 polyFn co_Count
  (fun u => coZ')
  (fun z => coZ')
  (fun (A : Set) (f : A → coN')
       (B : Set) (g : B → coN')
       (x : A × B) =>
    let (a, b) := x in
     add' (f a) (g b))
  (fun (A : Set) (f : A → coN')
       (B : Set) (g : B → coN')
       (x : A + B) =>
    match x with
    | inl a => f a
    | inr b => g b
    end).
```

We can use `co_count` to define count on lists:

```
Definition countList' :=
  specTerm' list_struct co_count ((list, tt), tt).

CoFixpoint countList (A : Set) (f : A → coN')
  (l : list A) : coN' :=
 countList' (countList, tt) f (fromList l).
```

4.6.5 Count Fusion

To be able to do proofs about `count`, we need a notion of bisimilarity on conatural numbers. Again, we will need two versions: `bisim_coN` and `bisim_coN'`, where `bisim_coN'` needs an additional constructor to deal with addition. We then need a proof that when two numbers of type `coN'` are related under the bisimilarity relation `bisim_coN'`, the flattening of these numbers will be related under the bisimilarity relation `bisim_coN`.

The generalization of the count fusion proof of Section 3.6 to the coinductive `co_count` is straightforward (Figure 4.3). Unfortunately, when we attempt to prove count fusion for lists in a similar manner to the proof of preservation of identity in Section 4.4, we find that Coq rejects the proof because of an unguarded corecursive call: the proof generated by proof specialization is more complicated than the proof we would do by hand, and the guardedness checker is not sophisticated enough to be able to see that the proof is indeed guarded. This is mostly due to the ubiquitous manipulation of heterogeneous equalities in the proofs generated by proof specialization; it is future work to see if we can either change proof specialization to return simpler proofs or make another modification to the guardedness checker to ensure that Coq is able to verify that the proof is in fact guarded.

To illustrate that the approach does work in principle, we can give a manual proof of count fusion for `list'`, and use this manual proof to construct a proof of count fusion for `list`, following the same structure as the proof for `mapIdList` in Section 4.4. These two proofs can be found in Figure 4.4.

The polytypic proof `co_countFusion` in Figure 4.4 is accepted by Coq as guarded. Modulo some matching on equalities, this proof corresponds very closely to the following corecursive definition:

```
CoFixpoint countFusionList (A : Set) (f g : A → coN')
  (h : coN' → coN') (Hmorph : morphism_coN' h)
  (Hx : ∀ h0 : coN' → coN', morphism_coN' h0 →
    ∀ x : A, bisim' (h0 (f x)) (g x))
  (l : list A)
  : bisim'_P l (h (countList f l)) (countList g l) :=
 countFusionList' countFusionList f g Hmorph Hx (fromList l)
```

and we see that the structure is again very similar to that of `mapList`.

```
Definition co_Fusion : PolyProp 1 2 co_Count:=
 polyProp 1 2 co_Count
  ((None, tt), ((None, tt), tt))
  (fun types fns =>
    let t := fst types in
    let f1 := fst fns in let f1' := snd fns in
    let f2 := fst f1' in
    ∀ h : coN' → coN', morphism_coN' h →
     ∀ x : t, bisim'_P 1 (h (f1 x)) (f2 x)).

Definition co_countFusion : PolyProof co_count co_Fusion.
Proof.
 apply (polyProof co_count co_Fusion);
 compute -[flatten_coN'].
(* unit *)
 intros h Hmorph u.
 destruct Hmorph as (Hzero, Hadd).
 apply bisim'_P_trans with (d1:=0) (d2:=0) (n2:=coZ'); auto.
 apply bisim'_bisim'_P; apply Hzero.
 apply bisim_coZ'_P.
(* int *)
 intros h Hmorph z.
 destruct Hmorph as (Hzero, Hadd).
 apply bisim'_P_trans with (d1:=0) (d2:=0) (n2:=coZ'); auto.
 apply bisim'_bisim'_P; apply Hzero.
 apply bisim_coZ'_P.
(* prod *)
 intros A fns Ha B fns' Hb h Hmorph p.
 destruct p as (a, b);
       destruct fns as (f1, f'); destruct fns' as (g1, g').
 destruct f' as (f2, _); destruct g' as (g2, _).
 destruct Hmorph as (Hzero, Hadd).
 apply bisim'_P_trans with (d1 := 0) (d2 := 0)
   (n2 := add' (h (f1 a)) (h (g1 b))); auto.
 apply bisim'_bisim'_P; apply Hadd.
 apply bisim_add'_P with (dm := 1%nat) (dn := 1%nat).
 apply Ha; auto.
 apply Hb; auto.
(* sum *)
 intros A fns Ha B fns' Hb h Hmorph s.
 destruct fns as (f1, f'); destruct fns' as (g1, g').
 destruct f' as (f2, _); destruct g' as (g2, _).
 destruct s; auto.
Defined.
```

Figure 4.3: Count Fusion, Coinductively

```
Lemma countFusionList'
 (rec : ∀ (A : Set) (f g : A → coN') (h : coN' → coN'),
   morphism_coN' h →
   (∀ h : coN' → coN', morphism_coN' h →
     ∀ x : A, bisim' (h (f x)) (g x)) →
   ∀ l : list A, bisim'_P 1 (h (countList f l)) (countList g l))
 (A : Set) (f g : A → coN') (h : coN' → coN')
 : morphism_coN' h →
   (∀ h : coN' → coN', morphism_coN' h →
     ∀ x : A, bisim' (h (f x)) (g x)) →
  ∀ l : list' A, bisim'_P 1 (h (countList' (countList, tt) f l))
                            (countList' (countList, tt) g l).
Proof.
 intros rec A f g h Hmorph Hx l.
 elim Hmorph; intros Hzero Hadd.
 destruct l; compute [countList' specTerm']; simpl.
 apply bisim'_P_trans with (d1:=0) (d2:=0) (n2:=coZ'); auto.
 apply bisim'_bisim'_P; apply Hzero.
 apply bisim_coZ'_P.
 destruct d as (a, l').
 apply bisim'_P_trans with (d1 := 0) (d2 := 0)
  (n2 := add' (h (f a)) (h (countList f l'))); auto.
 apply bisim'_bisim'_P; apply Hadd.
 apply bisim_add'_P with (dm := 0) (dn := 1%nat).
 apply bisim'_bisim'_P; apply Hx.
 apply Hmorph.
 apply (rec A f g h Hmorph Hx).
Defined.

Theorem countFusionList (A : Set) (f g : A → coN')
  : ∀ (h : coN' → coN'), morphism_coN' h →
    (∀ h : coN' → coN', morphism_coN' h →
      ∀ x : A, bisim' (h (f x)) (g x)) →
    ∀ l : list A,
  bisim'_P 1 (h (countList f l)) (countList g l).
Proof.
 cofix countFusionList; intros A f g h Hmorph Hx l.
 rewrite (decomp_id_coN' 1 (countList f l)); simpl.
 rewrite (decomp_id_coN' 1 (countList g l)); simpl.
 elim Hmorph; intros Hzero Hadd.
 change (bisim'_P 1 (h
  (decomp_coN' 1 (countList' (countList, tt) f (fromList l))))
  (decomp_coN' 1 (countList' (countList, tt) g (fromList l)))).
 do 2 rewrite <- decomp_id_coN'.
 apply
  (countFusionList' countFusionList f g Hmorph Hx (fromList l)).
Defined.
```

Figure 4.4: Count Fusion Specialized to `list`

Chapter 5

Related Work

As already explained in the Chapter 1, "generic programming" is a very general term which has been instantiated in many contexts. A detailed survey of generic programming approaches is therefore well beyond the scope of this thesis. For a detailed survey on generic programming in Haskell, we refer the reader to a survey paper by Hinze et al. (2006a), or the paper by Rodriguez et al. (2008) which gives an overview of the numerous "light-weight" generic programming approaches implemented as libraries in Haskell. Instead, we will restrict ourselves to datatype-generic programming, which for the purposes of this chapter we will roughly define as

Definition 1 (Datatype-generic programming) *A datatype-generic function is a single function that can be applied to terms of many different datatypes.*

In particular, type classes do *not* classify as datatype-generic programming under this definition because a type class is a collection of functions, one for each datatype; although we can abstract over this collection, type classes do not constitute a *single* function that can be applied to terms of different datatypes.

This definition still includes many different approaches, many of which are only marginally related to our work in this thesis. Most of this chapter will therefore have the following shape:

1. We start with the current focus (starting with our broad definition of datatype-generic programming, above)
2. We classify approaches that fit the current scope into two or more groups, only one of which is directly related to the approach we take in this thesis
3. We give a canonical example for each group that is *not* directly related
4. We narrow the scope to the group which includes the approach we take, and reiterate.

The chapter concludes with a brief discussion of research on productivity.

5.1 Scrap Your Boilerplate

Approaches to datatype-generic programming can be divided into two major groups: approaches that take advantage of certain properties of the types of the terms they are applied to, and approaches that do not. In this section we will discuss one well-known approach to generic programming that does not take advantage of properties of the datatype: *Scrap Your Boilerplate*. In the next section we will focus in on the second group.

The Scrap Your Boilerplate (SYB) approach (Lämmel and Jones, 2003) —and its numerous variations (Lämmel and Jones, 2005; Hinze et al., 2006b; Hinze and Löh, 2006b)— is aimed towards programs that traverse data structures built from mutually recursive datatypes. These programs usually consist of a lot of "boilerplate" code to traverse the structure and a small amount of "real" code that does the important work. The idea is that most of this boilerplate code can be written once and then reused, leaving the programmer free to concentrate on more important parts of the code. To achieve this a small library is provided containing two types of generic combinators: recursive traversals and type extensions. Generic functions are then defined in terms of these library functions. We will give a short explanation of how this approach works, using a function that increases the salary of every employee of a company as an example.

In this example, the boilerplate code would traverse the company structure, perhaps going through all departments, all units within these departments, all employees in these units and finally, the salary of these employees. Only when we have traversed through the entire company structure and found the employees will the algorithm actually do any useful work: increase the salary of these employees, given by the function `incSalary`.

```
data Salary = S Float
data Company = Company (List Dept)
data Dept = Dept Name (List SubUnit)
data SubUnit = EUnit Employee | DUnit Dept
...

incCompany :: Float -> Company -> Company
incCompany k (Company ds) = Company (map (incD k) ds)
incDept :: Float -> Dept -> Dept
incDept k (Dept nm us) = Dept nm (map (incUnit k) us)
...
incSalary :: Float -> Salary -> Salary
incSalary k (S s) = S (s × (1 + k))
```

Depending on the company structure, there could be a long list of traversals before finally the function `incSalary` is reached. Using the Scrap Your Boilerplate approach none of the traversal methods are necessary, and can be replaced by the following code:

```
increase :: Float -> Company -> Company
increase k = everywhere (mkT (incSalary k))
```

The function `mkT` transforms `incSalary` so that it can be applied to any node in the tree, not just `Salary` nodes; this is called type extension. The resulting function behaves like `incSalary` when applied to a `Salary` and like the identity function otherwise.

The function `everywhere` is a generic traversal combinator that applies its argument function to every node in the tree. In addition to this there are other traversal combinators that, for example, apply their argument functions to only those nodes in the tree that satisfy certain conditions.

The main advantage of this approach is that it can deal with arbitrary datatypes, including nested and mutually recursive datatypes, without any extra complications. However, the Scrap Your Boilerplate approach can be seen as orthogonal to the Generic Haskell approach: in SYB a small amount of boilerplate code is still necessary to define the generic traversals. This boilerplate might be eliminated using Generic Haskell.

To be able to compare the Scrap Your Boilerplate approach to other generic approaches, Hinze et al. (2006b) retrofit a generic view to the SYB approach: the *spine view*. Hinze and Löh (2006b) then identifies some limitations of the SYB/spine view approach and suggests improvements:

- The spine view is inherently value-oriented: any value can be regarded as a data constructor applied to other values. This introduces the limitation that we can only define generic functions that *consume* data, since we do not know which constructors together constitute a datatype. To remedy this Hinze and Löh (2006b) introduce the *typed spine view*. This view has access to additional information about datatypes; in particular, a datatype is viewed as a *list* of constructor applications, allowing us to write generic functions that produce data.

- The spine view is limited to datatypes of kind $\star$. Generic functions such as `map`, that abstract over type constructors are therefore out of reach. To address this problem, Hinze and Löh (2006b) again define a new view: the *lifted spine view*, which works in a very similar manner as the spine view, but with each constructor lifted to types of kind $\star \to \star$. However, the approach remains limited to datatypes of a finite set of kinds.

For a comprehensive overview of these different spine views, including mechanisms to combine them, see (Hinze and Löh, 2009), which is largely based on (Hinze and Löh, 2006a).

The use of type casts in the SYB approach complicates proving properties about generic functions in SYB. As far as we are aware the only work dealing with proofs about SYB functions (Reig, 2006) converts SYB programs into Generic Haskell first and then applies the techniques described in (Hinze, 2000a).

5.2 Origami Programming

We now start to narrow our definition slightly and consider only approaches to datatype-generic programming that take advantage of some properties of the datatype. Again, we can split these approaches into two groups: semantic versus syntactic approaches. While semantic approaches use properties of the type arguments to define generic functions—such as the fact that the datatype is a functor—it does not inspect the datatype. In contrast, in syntactic approaches the structure of the datatype is essential and generic functions are defined by induction on this structure. We will zoom in on syntactic approaches, which we will refer to as *polytypic programming*, in the next section. An example of a semantic approach is what is often referred to as *origami programming* because of the abundance of folds and unfolds. In this section we will discuss origami programming in some more detail.

Origami programming (Gibbons, 2006) is based on the idea that datatypes are fixpoints of pattern functors, where the pattern functor describes the shape of the datatype. The theory of datatypes as fixpoints of functors is explained in detail in (Backhouse et al., 1998). We can define a datatype `Fix`, which takes such a pattern functor s and a type a of kind $\star$, and gives us the fixpoint of f:

```
data Fix s a = In {out :: s a (Fix s a)}
```

The pattern functor for lists and, using that, the definition of the `List` datatype can be defined as follows:

```
data List° a b = NilF | ConsF a b
type List a = Fix List° a
```

Note that `List°` is not recursive, and takes an extra argument b which effectively represents the recursion. The type constructor `Fix` then ties the recursive knot in the definition of `List`.

There are some restrictions on the datatypes to which the type constructor `Fix` can be applied: the approach is limited to datatypes that can be represented by pattern functors which are bifunctors (which in Haskell terms restricts it to regular datatypes of kind $\star \to \star$), captured by the following type class:

```
class Bifunctor s where
 bimap :: (a → c) → (b → d) → (s a b → s c d)
```

where `bimap` should preserve identity and composition, although those properties cannot be expressed in Haskell. As another example of a pattern functor, consider the type of trees with elements only at its leaves:

```
data Tree° a b = LeafF a | NodeF b b
type Tree a = Fix Tree° a
```

Both `Tree°` and `List°` can be shown to be instances of the `Bifunctor` class:

```
instance Bifunctor List° where
```

```
 bimap f g NilF = NilF
 bimap f g (ConsF x y) = ConsF (f x) (g y)

instance Bifunctor Tree° where
 bimap f g (LeafF x) = LeafF (f x)
 bimap f g (NodeF t1 t2) = NodeF (g t1) (g t2)
```

Generic functions can now be defined in terms of `bimap` and the constructor and destructor for the datatype `Fix`:

```
map :: Bifunctor s => (a → b) → Fix s a → Fix s b
map f = In ∘ bimap f (map f) ∘ out

fold :: Bifunctor s => (s a b → b) → Fix s a → b
fold f = f ∘ bimap id (fold f) ∘ out
```

There is a trade-off in the range of datatypes covered by an approach and the elegance with which functions such as `fold` can be defined. By restricting the approach to datatypes whose pattern functors are bifunctors, we get a very short and elegant definition of `fold`; whereas in Generic Haskell the `fold` function requires some extra machinery, but there is no such restriction on the datatypes for which a generic function is defined.

The use of a type class to capture the notion of a bifunctor has the disadvantage that the user must provide an instance of `Bifunctor` for each new pattern functor (and thus each new datatype). The language extension PolyP, which we will discuss in Section 5.3.1, eliminates this restriction by not treating the datatype as a black box.

Proofs about generic origami functions can be done using the theory of initial algebras. See (Gibbons and Paterson, 2009) for some example properties and proofs that use the theory of natural transformations.

5.3 Polytypic Programming in Haskell

We now narrow our focus further and consider only approaches which take advantage of the structure of the datatype. We call such approaches *polytypic* and the approach that we use in this thesis is one example. There are, however, various possible views on the construction of datatypes. Examples of views include the "*sums of products*" view we take in this thesis and is essentially the view used in Generic Haskell and Generic Clean. Other examples include the view in PolyP, one of the earliest approaches to polytypic programming, and the *spine* view, a more recent development we discussed in Section 5.1.

The approaches we discuss in this section all have in common that polytypic functions are not first class: we cannot define functions that take polytypic functions as an argument, only specific instantiations of polytypic functions can be passed as arguments to other functions. In the next section, we will refine our focus one final time and concentrate on approaches to polytypic programming implemented in dependently typed languages, where polytypic functions become first-class.

5.3.1 PolyP

PolyP (Jansson and Jeuring, 1997) was the first generic language extension for Haskell. The approach taken in PolyP is in a sense a variation on origami programming, since it also treats datatypes as fixpoints of pattern functors—and is therefore also restricted to regular datatypes of kind $\star \to \star$. However, there is one important difference: PolyP allows us to inspect the structure of these pattern functors in order to define generic functions, thus allowing for different behaviour of the function for different datatypes and expanding the range of functions that can be defined generically.

Pattern functors are type constructors of kind $\star \to \star \to \star$ representing the recursive structure of a datatype, and they can be built using the following grammar:

$$f ::= g + h \mid g \times h \mid \mathit{Par} \mid \mathit{Rec} \mid d@g \mid \mathit{Const}\ t \mid \mathit{Empty}$$

Each functor is a sum of products of either a parameter (*Par*), the datatype itself (*Rec*), the composition of datatype d and functor g ($d@g$) or constant types ($\mathit{Const}\ t$, where t may be *Int* or *Char*, etc.). An empty product is represented by the unit type *Empty*.

By forcing a pattern functor to conform to this grammar we ensure that it is a bifunctor. Given the same type constructor `Fix` as in Section 5.2 (which we will repeat here for convenience), we can define the pattern functor for lists and apply `Fix` to get the type `List`:

```
data Fix s a = In {out :: s a (Fix s a)}

data List° = Empty + Par × Rec
type List a = Fix List° a
```

We should now be able to define the generic `fold` function in a similar fashion as we did for origami programming, except that we do not have an explicit `bimap` function as part of the `Bifunctor` class that we can use. However, we can define `bimap` polytypically by induction on the structure of pattern functors. This is a distinct advantage over origami programming because it eliminates the need to provide an instance of the type class for each new datatype. The polytypic `bimap` function is defined as follows:

```
polytypic bimap :: (a → c) → (b → d) → s a b → s c d
 = λ p r →
  case s of
   g + h    → (bimap p r) -+- (bimap p r)
   g × h    → (bimap p r) -×- (bimap p r)
   Empty   → id
   Par     → p
   Rec     → r
   d@g     → map (bimap p r)
   Const t → id

with
```

```
(-+-) :: (g A B → g C D) → (h A B → h C D) →
              ((g + h) A B → (g + h) C D)
(-×-) :: (g A B → g C D) → (h A B → h C D) →
              ((g × h) A B → (g × h) C D)
```

where `map` is the generic map function defined on bifunctors. Since we now have a polytypic function `bimap`, we can define fold generically in exactly the same way as we did in the origami programming approach in Section 5.2.

Due to the explicit recursion marker in pattern functors, functions such as `fold` can easily be defined in PolyP. At the same time this poses the restriction that generic functions in PolyP only act on regular datatypes of kind $\star \to \star$. It has proven difficult to define a view on datatypes that includes such an explicit recursion marker which is not restricted to regular datatypes. Rodriguez et al. (2009) make use of recent Haskell extensions such as generalized algebraic datatypes (GADTs) and type families to describe a technique that allows a fixpoint view for systems of mutually recursive (and thus non-regular) datatypes.

5.3.2 Generic Haskell

Since the approach that we take in this thesis was designed to model that used in Generic Haskell, we can be brief in this section. Generic Haskell (Löh, 2004) represents datatypes using the *sums of products* view: the choice of constructors is represented by a sum, whereas the list of arguments to a particular constructor is represented by a product with the nullary constructor represented by the unit type. In addition, Generic Haskell records information about constructors (`Con`) and record labels (`Label`) such as the name and fixity of the constructor or label. This information can be used in generic functions such as pretty printers, where the name of the constructor is needed.

In addition to the extra information stored about constructors and labels, an important difference between the structural types in Generic Haskell and PolyP (as discussed in Section 5.3.1) is the absence of an explicit recursion marker. This difference in structural types is in fact the most notable difference between Generic Haskell and PolyP and the most important other differences stem from this.

In Generic Haskell, the structural types for `List` a and `Tree` a b (trees with elements of type a at the leaves and elements of type b at the nodes) are given as:

```
type List° a =
        Con "Nil" (..) Unit
   :+: Con "Cons" (..) (a :×: (List a))
type Tree° a b =
        Con "Leaf" (..) a
   :+: Con "Branch" (..) (b :×: ((Tree a b) :×: (Tree a b)))
```

These structural types represent the top-level structure of a datatype. A type T and its structural representation T° are isomorphic, and this isomorphism is witnessed by an embedding-projection pair of type

```
data a ↔ b = EP {from :: a → b, to :: b → a}
```

The Generic Haskell compiler generates these embedding-projection pairs, and they are not available to the generic programmer (unlike in PolyP, where the application of `In` and `out` is the responsibility of the programmer).

A generic program is then defined by induction on these structural types. For example, the generic `map` function in Generic Haskell takes the form

```
map{|T :: k|} :: Map{[k]} T T
map{|Unit|} Unit = Unit
map{|Int|} i = i
map{|Char|} c = c
map{|:+:|} mA mB (Inl x) = Inl (mA x)
map{|:+:|} mA mB (Inr x) = Inr (mB x)
map{|:×:|} mA mB (x :×: y) = mA x :×: mB y
map{|Con c|} m (Con x) = Con (m x)
map{|Label l|} m (Label x) = Label (m x)
```

Note that the type of the generic `map` function is given by a kind-indexed type `Map` as we have discussed in Section 2.4.1.

Given this generic definition of `map` it can be specialized to any type that can be described by the structural representation in Generic Haskell. This specialization is done at compile time and is hidden from the programmer.

Generic Clean is very similar to Generic Haskell; the main difference is the integration of generics with the type class system, which has some advantages (Alimarine and Plasmeijer, 2001). Moreover, a lot of effort is put into reducing the efficiency overhead of polytypic programming (Alimarine and Smetsers, 2004, 2005).

5.4 Dependent Polytypic Programming

PolyP, Generic Haskell and Generic Clean all rely on preprocessors or language extensions to define generic programming. As a consequence, polytypic functions are not first-class; in addition, the language extension or preprocessor must include an extension of the host type system to typecheck polytypic functions (if such an extension is absent, only the *instances* of the polytypic functions can be typechecked).[1]

When we move to a language with a much more powerful *dependent* type system we can do polytypic programming entirely within the host language. Polytypic functions can now be made ordinary terms in the host language and are therefore first-class. The type checker for the host language can typecheck the polytypic functions since we can define what it means for a polytypic function to be well-typed *within* the host language.

We have now arrived at the class of related work which is most closely related to ours, and we discuss a number of approaches in this section.

[1]The spine view lives in between the two worlds as it uses generalized algebraic datatypes: a poor man's implementation of dependent types. Polytypic functions still are not first-class, however.

5.4.1 PolyP-style Generics

The work by Pfeifer and Rueß (1999) has been cited (Benke et al., 2003) as the first formalization of polytypic programming in a dependent language. The paper consists of two parts. The first part of the paper formalizes the semantic bifunctor-based approach that we know from origami programming (Section 5.2). The translation to a dependently typed language (in fact, Coq) is reasonably straightforward and we do not reproduce it here. However, note that definitions such as `map` and `fold` as given in Section 5.2 *cannot* be given, as the recursion is not structural (see also Chapter 4). Instead, it is assumed that as part of the proof of the initiality of a datatype a dependent eliminator (i.e., a fold) is *given* (Definitions 9 and 10 in their paper). The polytypic function `map` can then of course easily be defined.

The second part of the paper formalizes the syntactic approach familiar from PolyP. The universe that is considered is simpler than the universe in PolyP, however, and is defined as shown in Figure 5.1 (we have paraphrased the definition to make it easier to compare to the other definitions in this chapter).

Like in PolyP, the syntactic approach is used to define `bimap` once and for all for all datatypes that can be represented in the universe. Moreover, the bimap laws (preservation of identity and composition) are proven. Although this is clearly a proof about a polytypic function, it is not a "polytypic" proof in the sense that the proof is by direct induction on the universe: no special "proof specialization" machinery is in place. It is also an extremely simple proof; to emphasize this, we have shown the full proof in the figure.

Note that here (like in PolyP) the bifunctor is not recursive (and does not need to be): the recursion is handled by an explicit parameter. Moreover, the authors do *not* give a definition of the initial algebra induced by these (bi)functors: in other words, recursion is not handled polytypically.

Benke et al. (2003) continue from this work; although it seems that they *do* give a decoder for the universe and (for instance) give a fold operation, well-definedness (termination) of the fold operation is taken for granted. For their simple universe (the first of a series), they define fold as

$$\texttt{iter}_\Sigma\ C\ d\ (\texttt{Intro}_\Sigma\ x) = d\ (F^1_\Sigma\ (\texttt{iter}_\Sigma\ C\ d)\ x)$$

Here, d is the "step" function of the fold; the definition maps (F^1_Σ) the fold across all recursive occurrences of the datatype and then applies the step function to the result. This is the usual definition of a fold operation, but note that the recursive occurrence of $\texttt{iter}_\Sigma$ is not (obviously) applied to structurally smaller arguments. In fact, it is not applied to an argument at all. Of course, since F^1_Σ will only apply its argument to the recursive occurrences of the term, and since those occurrences must be structurally smaller since the term is inductive, this definition *is* well-defined. However, in a type theory such as Coq that relies on "obvious" structural induction, a definition such as this will not be accepted (if it was, we would not have any of the difficulties we describe in Chapter 4).

```
Inductive code :=
  | sum : code → code → code
  | prod : code → code → code
  | var : code
  | rec : code.

Fixpoint Bifunctor (c : code) (A X : Set) : Set :=
  match c with
  | sum c1 c2 => Bifunctor c1 A X + Bifunctor c2 A X
  | prod c1 c2 => Bifunctor c1 A X × Bifunctor c2 A X
  | var => A
  | rec => X
  end.

Fixpoint bifunctor (c : code) (A B X Y : Set)
                   (f : A → B) (g : X → Y)
  : Bifunctor c A X → Bifunctor c B Y :=
  match c return Bifunctor c A X → Bifunctor c B Y with
  | sum c1 c2  => fun s =>
      match s with
      | inl l => inl (bifunctor c1 f g l)
      | inr r => inr (bifunctor c2 f g r)
      end
  | prod c1 c2 => fun p =>
      let (l, r) := p
      in (bifunctor c1 f g l, bifunctor c2 f g r)
  | var => fun a => f a
  | rec => fun x => g x
  end.

Lemma bifunctor_id : ∀ (c : code) (A X : Set)
                       (e : Bifunctor c A X),
  bifunctor c (fun x => x) (fun x => x) e = e.
Proof.
  induction c; simpl; intros; auto.
  destruct e; [rewrite IHc1 | rewrite IHc2]; reflexivity.
  destruct e; rewrite IHc1; rewrite IHc2; reflexivity.
Qed.

Lemma bifunctor_comp : ∀ (c : code) (A B C X Y Z : Set)
                         (f2 : B → C) (f1 : A → B)
                         (g2 : Y → Z) (g1 : X → Y)
                         (e : Bifunctor c A X),
  bifunctor c f2 g2 (bifunctor c f1 g1 e) =
  bifunctor c (fun a => f2 (f1 a)) (fun x => g2 (g1 x)) e.
Proof.
  induction c; simpl; intros; auto.
  destruct e; [rewrite IHc1 | rewrite IHc2]; reflexivity.
  destruct e; rewrite IHc1; rewrite IHc2; reflexivity.
Qed.
```

Figure 5.1: Universe in (Pfeifer and Rueß, 1999)

The authors continue by extending the simple universe in various ways: first to (almost) the universe of PolyP and then beyond by supporting mutually recursive datatypes as well as indexed datatypes. They give a proof of reflexivity and substitutivity of Boolean equality in the smaller universe. The definition and the proof are defined in terms of the fold operator that follows from the initial algebra so this can really be regarded as a "polytypic proof".

Compared to the universe we consider in this thesis, however, the class of kinds supported by both is limited to first-order kinds (in the case of (Pfeifer and Rueß, 1999), even only kind $\star \to \star$, like in PolyP).

5.4.2 Strictly Positive Types

In (Morris et al., 2006) we finally find a way to decode universes with recursion. The approach relies on an cunning idea: rather than giving a direct translation from a code to a datatype, the authors give a datatype which is *indexed* by a code: rather than giving the translation to a datatype, they formalize what it means for a datatype corresponding to a code to be inhabited. Before we consider the full universe of strictly positive types, we first give a Coq-formalization of a "Morris-style" decoder for the simple Pfeifer universe from the previous section in Figure 5.2.

The Pfeifer universe contains two distinguished free variables, `var` and `rec`. In the definition of the bifunctor in Section 5.4.1 we treated both variables simply as free variables, but now we want to treat `rec` as marking the recursive occurrences of the datatype. Nevertheless, as in any decoder, we need an environment that decodes the free variables in the code: that is, we need the decoding of `var` and the decoding of `rec`. Since the length of this environment is fixed, we have simply inlined it as two arguments to the decoder `El`: the decoding of `var` is a type A, but the decoding of `rec` is a *code* `Rec`. This is crucial to be able to give an interpretation of recursion: the decoding of recursion is simply the decoding of the entire code again. We will discuss if this approach can be used for Generic Haskell-style generics in Section 6.4.2.

Morris et al. (2006, 2007a) give a universe of strictly positive types, shown in Figure 5.3. The syntax in the figure is that of Epigram, but is hopefully self-explanatory. It defines a dependent datatype `SPT`, indexed by a natural number n which indicates the number of free variables in the type. The constructors are given in natural deduction style, with the arguments to the constructor above the line and the result type of the constructor below.

When we compare this universe to the universe we use in this thesis (Figure 2.4) we find many differences. Some seem merely cosmetic; for example, the type constants and the "0" and "successor" constructors for selecting free variables have been inlined into the definition of the universe.[2] More important differences come from the two constructors

[2]The authors say that inlining the constructors for selecting free variables makes proofs easier; it would be interesting to see if it would have the same effect in our universe.

```
                     the environment
Inductive El' (A : Set) (Rec :  code) : code → Set :=
  | El_void :
      El' A Rec one
  | El_inl : ∀ (l r : code),
      El' A Rec l →
      El' A Rec (sum l r)
  | El_inr : ∀ (l r : code),
      El' A Rec r →
      El' A Rec (sum l r)
  | El_pair : ∀ (l r : code),
      El' A Rec l →
      El' A Rec r →
      El' A Rec (prod l r)
  | El_top :
      A →
      El' A Rec var
  | El_in :
      El' A Rec Rec →
      El' A Rec rec.

(* Decoding "closed" types *)
Definition El (A : Set) (c : code) : Set := El' A c rec.

(* Pattern functor for lists *)
Definition ListF : code := sum one (prod var rec).

(* Embedding lists into the structural representation *)
Fixpoint fromList (A : Set) (xs : list A) : El A ListF :=
  match xs with
  | nil =>
      El_in (El_inl El_void)
  | cons x xs' =>
      El_in (El_inr (El_pair (El_top x) (fromList xs')))
  end.
```

Figure 5.2: Morris-style Decoder for the Pfeifer Universe

$$\text{data}\quad \frac{n : \mathtt{Nat}}{\mathtt{SPT}\ n : \star} \quad\text{where}\quad \frac{}{\mathtt{vz} : \mathtt{SPT}\ (\mathtt{s}\ n)} \quad \frac{T : \mathtt{SPT}\ n}{\mathtt{vs}\ T : \mathtt{SPT}\ (\mathtt{s}\ n)}$$

$$\frac{}{\text{‘0’} : \mathtt{SPT}\ n} \quad \frac{}{\text{‘1’} : \mathtt{SPT}\ n} \quad \frac{S, T : \mathtt{SPT}\ n}{S\text{‘}{+}\text{’}T : \mathtt{SPT}\ n} \quad \frac{S, T : \mathtt{SPT}\ n}{S\text{‘}{\times}\text{’}T : \mathtt{SPT}\ n}$$

$$\frac{K : \star \qquad T : \mathtt{SPT}\ n}{K\text{‘}{\rightarrow}\text{’}T : \mathtt{SPT}\ n} \quad \frac{F : \mathtt{SPT}\ (\mathtt{s}\ n)}{\text{‘}\mu\text{’}F : \mathtt{SPT}\ n}$$

Figure 5.3: Universe of Strictly Positive Types

```
gMap φ x ⇐ rec x
  gMap (mF φ f) (top x)     ⇒ top (f x)
  gMap (mU φ)   (top x)     ⇒ top (gMap φ x)
  gMap ml       (top x)     ⇒ top x
  gMap (mF φ f) (pop x)     ⇒ pop (gMap φ x)
  gMap ml       (pop x)     ⇒ pop x
  gMap (mU φ)   (pop x)     ⇒ pop (gMap φ x)
  gMap φ        (in x)      ⇒ in (gMap (mU φ) x)
  gMap φ        void        ⇒ void
  gMap φ        (inl x)     ⇒ inl (gMap φ x)
  gMap φ        (inr x)     ⇒ inr (gMap φ x)
  gMap φ        (pair x y)  ⇒ pair (gMap φ x) (gMap φ y)
  gMap φ        (fun f)     ⇒ fun (λk ⇒ gMap φ (f k))
```

Figure 5.4: Generic Map in (Morris et al., 2007a)

on the bottom line in the figure: the type constants include the function space (but the domain of the function must be a constant type) and the universe includes recursion. Their universe does not include type application, however, and therefore covers parametric types of first order kind.

Polytypic functions are functions on elements of the decoder datatype `El`. For example, Figure 5.4 shows the generic map function as given by Morris et al.. We have shaded the part that deals with free variables and the recursion markers; this part can probably be dealt with in a general way (just like we deal with variables in a general way in this thesis) and individual polytypic functions can ignore them. The non-shaded part contains the definitions for unit, sums, products, and arrows; these are very similar to their Generic Haskell counterpart, with the exception that the number of function arguments we get does not depend on the kind of the type constant. Rather, we always get an environment ϕ which contains functions for each of the free variables in the datatype.

In the paper, the authors prove the two functor laws "by easy induction". So it seems that proofs in their universe do not need the sort of infrastructure we have given in Chapter 3. It is not obvious what properties of their universe makes this possible, and it would be interesting to see if we can "backport" some properties of their approach to make our proofs easier. On the other hand, the definition of the functor laws is not as direct (for example, a special composition operator needs to be defined for composition of morphisms over environments).

Morris et al. (2007a) continue with an extension of the universe to include dependent datatypes, but a discussion of these "strictly positive families" is beyond the scope of this thesis.

5.4.3 Generic Haskell-style Generics

There are a few implementations of Generic-Haskell style generics in a dependent programming language: Altenkirch and McBride (2003) give an implementation in Oleg (the dependent language developed by Conor McBride in his doctoral thesis, McBride, 1999) and Norell (2002) presents a similar design in Alfa and follows (a preprint of) the first paper closely; Sheard (2007) goes some way towards a design in Ωmega. None of these formalizations attempt to do any proofs (polytypic or otherwise) over polytypic programs.

Since both our implementation in Coq in Chapter 2 and the implementations in Oleg and Alfa are based on the work by Hinze, it should come as no surprise that there are many similarities between both formalizations. Of course, the host language is different, which inevitably leads to variation in the design. Most notably, Oleg and Alfa appear to support general recursion, which simplifies the implementation of polytypic programs but is less suitable to the implementation of polytypic proofs.[3]

An important difference between these designs and our own is that the concept of a polytypic function is not reified in the host language: there is no data structure that corresponds to our `PolyFn` record (Section 2.4.2). Instead, polytypic functions are written by direct induction on the universe and there is no separate specialization process. We think that it is important to identify polytypic functions (and polytypic proofs) as stand-alone concepts, as we feel it makes functions and proofs easier to write. Moreover, since it forces polytypic functions to be more uniform (as most of the work is done by specialization, which is the same for every polytypic function), polytypic proofs can also be smaller. Finally, it will make the formalization more accessible to programmers used to Generic Haskell.

5.4.4 Containers

Containers (Hoogendijk and de Moor, 2000; Abott et al., 2003) and also (Jay, 1995; Backhouse and Hoogendijk, 2003) are a very different view of datatypes. Unfortunately, most of the literature on containers relies heavily on category theory and is therefore not easily accessible to people not well-versed in this subject. Section 5 of (Altenkirch et al., 2007) is one notable exception and our exposition here is mostly based on that paper.

The basic idea of container types is to separate out the shape of a term (like a list or a tree) from the values in the term. To interpret a term as a container, we need three components: we need a characterization of the *shape* of the term, we need a mapping from this shape to a set of *positions* within the term, and we need a mapping from these positions to the values in the term.

For instance, consider the type of vectors or lists of a given length (here and elsewhere in this section we will use Coq for our examples):

[3](Norell, 2002, Section 7.6, *Converting to real types*) goes as far as to consider the lack of recursion in type theory a “flaw” in type theory for the purposes of doing polytypic programming.

```
Inductive Vec (A : Set) : nat → Set :=
  | vnil : Vec A 0
  | vcons : ∀ n, A → Vec A n → Vec A (S n).
```

We can consider the shape of a vector to be its length, that is, a natural number. Given the shape of a vector, we can compute the set of *positions* in the vector. In this case, the set of positions is the set of indices $0 \ldots n-1$, which is computed by `Fin`:

```
Inductive Fin : nat → Set :=
  | fz : ∀ n, Fin (S n)
  | fs : ∀ n, Fin n → Fin (S n).
```

Finally, we can define a (total) mapping from this set of positions to the values of the elements in the vector:[4]

```
nth : ∀ (A : Set) (n : nat), Vec A n → Fin n → A.
```

A (unary) container then is a type S describing shapes and a mapping P from shapes to a type describing positions:

```
Inductive UCont : Type :=
  | ucont : ∀ (S : Set) (P : S → Set), UCont.
```

An *instance* of a container is given by a particular shape s and a mapping f from the positions in s to the values in the datatype:

```
Inductive UExt : UCont → Set → Type :=
  | uext : ∀ (S : Set) (P : S → Set) (X : Set)
             (s : S) (f : P s → X),
      UExt (ucont S P) X.
```

For instance, we can regard a vector as a container as follows:

```
Definition Vec_UExt (A : Set) (n : nat) (xs : Vec A n)
    : UExt (ucont nat Fin) A :=
  uext n (nth xs).
```

As another example, consider the type of binary trees with data at the branches (but none at the leaves). The shape of such a tree is itself a tree without any information about the values in the tree. We can describe these "tree shapes" as

```
Inductive TreeShape : Set :=
  | sleaf : TreeShape
  | snode : TreeShape → TreeShape → TreeShape.
```

Note that a natural number can similarly be considered as a list without values, with 0 taking the role of the empty list and successor taking the role of the `cons` constructor. Like the type of vectors was indexed by its shape (a natural number), the type of trees is indexed by the shape of trees:

[4]For conciseness we leave some of the definitions in this section as an (easy) exercise to the reader.

```
Inductive Tree (A : Set) : TreeShape → Set :=
  | tleaf :
      Tree A sleaf
  | tnode : ∀ (lsh rsh : TreeShape),
      A →
      Tree A lsh →
      Tree A rsh →
      Tree A (snode lsh rsh).
```

Next, we need a mapping from a shape to a set of positions: in this case, paths through a tree. For vectors, we had that both the type of vectors and the type of positions is indexed by the shape of the vector; here, both the type of trees and the type of paths through trees is indexed by a tree shape:

```
Inductive Path : TreeShape → Set :=
  | phere : ∀ (lsh rsh : TreeShape),
      Path (snode lsh rsh)
  | pleft : ∀ (lsh rsh : TreeShape),
      Path lsh →
      Path (snode lsh rsh)
  | pright : ∀ (lsh rsh : TreeShape),
      Path rsh →
      Path (snode lsh rsh).
```

Finally, we need a total mapping from positions to values in the tree (the equivalent of `nth` for vectors). This mapping is easily defined; here, we give its type only:

```
find : ∀ (A : Set) (s : TreeShape), Tree A s → Path s → A.
```

The treatment of both datatypes is very similar, and indeed we can now interpret trees as containers in a similar way that we interpreted vectors as containers:

```
Definition Tree_UExt (A : Set) (s : TreeShape) (t : Tree A s)
    : UExt (ucont TreeShape Path) A := uext s (find t).
```

Some definitions and proofs about containers can be done semantically. For instance, we can very easily prove (up to eta-expansion) that every container is a functor:

```
Definition ucmap (C : UCont) (X Y : Set) (f : X → Y)
                 (c : UExt C X) : UExt C Y :=
  match c in UExt C X return (X → Y) → UExt C Y with
  | uext _ _ _ s g => fun f => uext s (fun x => f (g x))
  end f.

Lemma ucmap_id : ∀ (C : UCont) (X : Set) (c : UExt C X),
  ucmap (fun x => x) c = c.

Lemma ucmap_comp : ∀ (C : UCont) (X Y Z : Set)
                     (f : Y → Z) (g : X → Y) (c : UExt C X),
  ucmap f (ucmap g c) = ucmap (fun x => f (g x)) c.
```

However, since containers can be built from “constant containers”, “sum containers”, “product containers”, etc. we can also give syntactic definitions and proofs by induction on the structure of the container. This makes it possible to define a generic equality function, for instance. Containers do not generalize easily to higher-order kinds (Conor McBride, personal communication), but do generalize to indexed (dependent) datatypes (Morris and Altenkirch, 2009). A discussion of indexed containers is, however, beyond the scope of this thesis.

5.5 Dealing with Termination and Productivity

Semantic approaches to dealing with termination are fairly well-established. The standard reference on Coq (Bertot and Castéran, 2004) discusses the use of a general accessibility predicate to define well-founded inductive functions (Section 15.2) and the use of domain specific accessibility (Section 15.4) as advocated by Ana Bove in her thesis (Bove, 2002).

Unfortunately, a similar approach to coinduction has not yet emerged and this is a current hot topic in type theory research. As discussed in Chapter 4, a coinductive function in Coq must satisfy a syntactic guardedness condition which informally guarantees that

1. Each corecursive call is made under at least one constructor.
2. If the corecursive call is under a constructor, it does not appear as an argument of any function.

However, there are many useful corecursive functions that do not satisfy this condition. The standard example of a function violating the first condition is the function which filters out the elements that do not satisfy some predicate P from a stream:

$$\text{filter } (x :: xs) = \begin{cases} x :: \text{filter } xs & \text{if } P(x) \\ \text{filter } xs & \text{otherwise} \end{cases}$$

In fact, the filter function is not guaranteed to be productive: if no element in the stream satisfies P then filter will never return an element and is therefore not productive. In (Bertot, 2005; Bertot and Komendantskaya, 2008) it is shown how to define functions such as filter by decomposing the problem into an inductive component (an inductive proof that there is at least one element satisfying P) and a coinductive component (a proof that this holds for all substreams of the stream).

A simple example of a function which fails to satisfy the second guardedness condition but is nevertheless productive is the function that defines the stream of Fibonacci numbers:

$$\text{fibs} = 0 :: 1 :: \text{fibs} \oplus \text{tail(fibs)}$$

where $\oplus$ is the pointwise addition of two streams. There are various ways in which functions such as these can be defined. One solution suggested by Danielsson and Altenkirch (2009) is to define an *ad-hoc* domain specific language: in this case, a coinductive datatype with a (coinductive or inductive) constructor for $\oplus$. We can then separately give an evaluation of the domain specific language to pure streams. Unfortunately, Coq does not currently support mixing inductive and coinductive definitions so that although the approach advocated by Danielsson and Altenkirch can be used in Coq, it requires an encoding of induction within a coinductive datatype.

Another possibility is given by Gianantonio and Miculan (2003), who give a theory of complete ordered families of equivalences which can be used to define functions with an inductive and a coinductive component. They give a definition of the infinite list of primes. This is an interesting example because it is effectively a constructive proof that there are infinitely many primes (in particular, the next larger prime number can always be generated in finite time).

Bertot and Komendantskaya (2009) suggest to avoid the direct use of coinduction and use induction only; for example, a stream can be defined as a function from natural numbers. Finally, Niqui (2009) uses notions from category theory to define a coinductive fixed point operator for a large class of coinductive functions (a "λ-coiteration scheme"), including the Fibonacci numbers.

Finally, there is some work in tracking termination and productivity through the type system instead (Abel, 2006). This is interesting as it frees us from syntactic restrictions, but unfortunately currently none of the major proof checkers implement it. Abel (2009) shows that the theory of sized types can be extended to Generic Haskell-style polytypic functions.

Chapter 6

Future Work and Conclusions

We will start this final chapter with an overview of future work. There is some scope for automation both in the definition of polytypic proofs and in the machinery required for recursion (Section 6.1). The use of syntactic sugar in a number of areas could be beneficial to hide some of the more complicated details from the user and make our definitions as intuitive as possible (Section 6.2). In Section 6.3 we discuss a number of useful extensions to the work in this thesis. To make use of the fact that polytypic functions in our development are higher-order, we need to slightly modify our definition of the `PolyFn` Record type. We would also like to extend the definition of polytypic properties to allow for properties about multiple polytypic function. Other extensions that might be of interest are type-indexed types, the extraction of code to Generic Haskell and the addition of some meta-information to our type universe to allow for such polytypic functions as pretty printers and parsers. We discuss the problem of recursion in Section 6.4, and in Section 6.5 we conclude the thesis.

6.1 Automation

6.1.1 Tactics for Polytypic Proofs

When we look closely at the proofs from Sections 3.2.3 and 3.6 we find a great deal of commonality. For example, the case for products in the proof that map preserves identities and the proof that equality is commutative are

$$\frac{\forall xy : A, f\ x\ y = f\ y\ x \qquad \forall xy : B, g\ x\ y = g\ y\ x}{f\ a\ a' \wedge g\ b\ b' = f\ a'\ a \wedge g\ b'\ b} \qquad \frac{\forall x : A, f\ x = x \qquad \forall x : B, g\ x = x}{(f\ a, g\ b) = (a, b)}$$

and the corresponding case in the proof of count fusion is

$$\frac{\begin{array}{l}\forall i\ j : \texttt{nat}, h\ (i+j) = h\ i + h\ j \\ \forall x : A, h\ (f\ x) = f'\ x \\ \forall x : B, h\ (g\ x) = g'\ x\end{array}}{h\ (f\ a + g\ b) = f'\ a + g'\ b}$$

All of these proofs are solved in a similar way (although the proof of count fusion requires one additional step). It should therefore be possible to construct such polytypic proofs almost fully automatically. There might be some base cases, like commutativity of equality on integers, which rely on library lemmas that need special treatment. However, the use of domain-specific tactics or hint databases should be sufficient to solve most of these special cases.

6.1.2 Generate Support for Recursion

At the moment quite a bit of machinery is required to specialize a polytypic function (or proof) to a coinductive type:

1. A structural representation type (`list'`) for every coinductive type (`list`).
2. The corresponding embedding-projection pair (translating between `list` and `list'`).
3. For each polytypic function (`map`) and each coinductive datatype (`list`) the corecursive definition (`mapList`), making use of the embedding-projection pair:
   ```
   CoFixpoint mapList (A B : Set) (f : A → B) (l : list A)
     : list B :=
    toList (mapList' mapList f (fromList l)).
   ```
4. Lemmas that the embedding-projection pair forms the witness of an isomorphism.
5. A definition of decomposition of terms of the coinductive type and a lemma that decomposition is an identity function.

Most of this infrastructure (excluding only the lemmas) is also needed in Generic Haskell, where it is generated by the Generic Haskell preprocessor. It would be useful to have tactics that can automatically generate these definitions.

The same infrastructure will be required for polytypic proofs, but here the situation is slightly trickier. For instance, the definition that map preserves identity, specialized to coinductive lists, is approximately

```
CoFixpoint mapIdList (A : Set) (f : A → A)
  (Hx : ∀ x : A, f x = x) (l : list A)
  : bisim_list (mapList f l) l :=
 to_preserves_bisim_list
   (mapIdList' mapIdList f Hx (fromList l)).
```

which follows the same structure as `mapList`, with `to_preserves_bisim_list` taking the role of one half of the embedding-projection pair for bisimilarity. However, the actual proof is more involved as it requires reasoning about equality (Section 4.4). In particular, it needs to make use of a decomposition lemma to unroll corecursive definitions.

6.2 Syntactic Sugar

Some of the definitions are a little cumbersome and could use some syntactic sugar to lighten the burden on the programmer. Ideally, we would like polytypic functions and types to be defined in Generic Haskell syntax. As is, we have put some effort into making the style of polytypic function definitions closely resemble the style used in Generic Haskell by carefully choosing our generic view on datatypes to match the view used in Generic Haskell. However, some additional syntactic sugar might ease the switch from one language to another.

A more pressing need for syntactic sugar can be found in the definition of polytypic properties. In this thesis we have mostly applied this syntactic sugar already, defining the property that `map` preserves identities (for kind $\star$) as

```
fun T f => ∀ x : T, f x = x
```

whereas the actual definition takes the form

```
fun types fns =>
 let T := fst types in
 let f := fst fns in
  ∀ x : T, f x = x
```

The arguments always take the form of tuples of types and functions and the user must extract the individual elements from the tuples to be able to use them in the definition of the property.

Another use for syntactic sugar can be found in the way we index tuples and represent variables: in both cases we use the `index` type. An element of type index is either `None` or takes the form `Some (Some (... None))`. It should be possible to use the standard syntax for natural numbers instead.

6.3 Extensions

6.3.1 Expressivity of Polytypic Properties

As defined in this thesis (Section 3.2.2), the type of a polytypic property generally looks like:

$$\forall(T_1,\ldots,T_{nt})\,.\,\texttt{Pt}\langle k\rangle\ \underset{1}{\underbrace{(T_1,\ldots,T_{nt})}} \times\cdots\times \texttt{Pt}\langle k\rangle\ \underset{nx}{\underbrace{(T_1,\ldots,T_{nt})}} \to \texttt{Prop}$$

This limits the expressiveness of polytypic properties, as they can only refer to a single polytypic type `Pt`. There are many properties that apply to a combination of polytypic functions. For example, we might want to express the property that mapping across a structure does not affect the count of that structure. This property would have a type similar to

$$\forall T_1\ T_2\ .\ \texttt{Count}\langle k\rangle\ T_2 \to \texttt{Map}\langle k\rangle\ T_1\ T_2 \to \texttt{Count}\langle k\rangle\ T_1 \to \texttt{Prop}$$

and might be defined as

```
fun T1 T2 f1 f2 f3 => ∀ x : T1, f1 (f2 x) = f3 x.
```

As another example, consider polytypic functions for encoding and decoding a term into a bitstream (Hinze and Jeuring, 2003, Section 1.4). A natural property we might want to state and prove is that decoding is right inverse to encoding. In (Jansson and Jeuring, 2002) many more examples of conversion functions and their inverses can be found.

The main difficulty in modifying properties in this way is the construction of the type arguments for each of the polytypic types. In Section 3.2.2 we have explained that

$$\underset{\sim\sim\sim\sim\sim\sim}{(T_1, \ldots, T_{nt})}_i$$

takes the correct *np* type arguments for the *i*th occurrence of `Pt` from the tuple of type arguments associated with the property. If we allow for different polytypic types we also get a different number of type arguments (*np*) for each of these polytypic types.

Since many of the lemmas concerning property and proof specialization involve these “shuffled tuples”, the lack of uniformity is likely to cause some difficulties, especially since those lemmas make heavy use of heterogeneous equality and are therefore rather complicated and difficult to modify. If this difficulty can be resolved we think the modification to allow properties over more than one polytypic function should be reasonably straight-forward.

6.3.2 Meta-information

In both Generic Clean and Generic Haskell the generic view on datatypes is extended with some meta-information like names of constructors. Such information is essential in the definition of certain polytypic functions such as pretty printers and parsers (Hinze and Jeuring, 2003, Section 2.5) and in larger scale applications of polytypic programming such as the automatic generation of graphical user interfaces for web elements (Plasmeijer et al., 2007). If we want to be able to reason about such functions, this meta-information must also be added to our universe. Although this may complicate the polytypic proofs, it should not pose any difficulties in the polytypic metatheory.

6.3.3 Extraction to Generic Haskell

The Coq proof assistant supports extraction of Coq programs to Haskell. This is important since it eliminates the gap between the system which has been proven correct and the system that is actually deployed. Since we have based our development on Generic Haskell, it would be useful if we could expand this extraction mechanism in such a way that polytypic functions in our system can be extracted to Generic Haskell code. Of course, the gap can never be completely eliminated. Proofs about polytypic functions are correct with respect to the specialization process that *we* have given. It is not guaranteed that they will still hold when using the specialization process used by Generic Haskell. However, this problem already exists without polytypicity: when extracting code to Haskell, it is assumed that the execution of that code is compatible with the execution of the code within Coq. It is not too much of a leap to make the same assumption about the Generic Haskell specializer.

Extraction to Generic Haskell should not be too difficult to implement. The extractor code must be able to recognize our `PolyFn` structure and generate the appropriate Generic Haskell code. It must also recognize which instances of the polytypic functions are used and generate the corresponding specializations. Of course, since the implementation of polytypic functions in a dependent language is strictly more expressive, not all Coq programs can be so translated. For example, no translation can be given for higher-order polytypic functions.

6.3.4 Type-indexed Types

Throughout this thesis we have used the polytypic map function as a running example. Polytypic map makes it possible to prove that a type on the universe is a functor if it is accompanied by the two corresponding functor laws: preservation of identity and composition. When we first introduced these properties (Section 3.2.1), we stated preservation of identity as

$$\begin{aligned}&\mathtt{Id}\langle\star\rangle\ T : (T \to T) \to \mathtt{Prop}\\&\mathtt{Id}\langle\star\rangle\ T = \lambda f : T \to T\ .\ \forall x : T\ .\ f\ x = x\end{aligned}$$

However, this lemma is too strong when we generalize to coinductive datatypes: we should not require *equality*, but *bisimilarity*. Unfortunately, although it is possible to give a definition of equality that can be used to compare terms of arbitrary types, there is no such definition for bisimilarity. Therefore, the property would have to be something like

$$\mathtt{Id}\langle\star\rangle\ T = \lambda f\ .\ \forall x : T\ .\ f\ x \approx_T x$$

where $(\approx)$ is a function $\forall (T : \star), T \to T \to \mathtt{Prop}$ for some type T. Unfortunately, such a function cannot be given for arbitrary types, but we *can* give such a function

```
Definition Bisim : PolyType 1 :=
 polyType 1 (fun A => A → A → Prop).

Definition bisim : PolyFn Bisim :=
 polyFn Bisim
   (fun x y : unit => True)
   (fun x y : tint => x = y)
   (fun (A : Set) (HA : A → A → Prop)
        (B : Set) (HB : B → B → Prop)
        ((a, b) (a', b') : A × B) =>
     HA a a' /\ HB b b'
   (fun (A : Set) (HA : A → A → Prop)
        (B : Set) (HB : B → B → Prop)
        (x y : A + B) =>
     match (x, y) with
     | (inl a, inl a') => HA a a'
     | (inr b, inr b') => HB b b'
     | _ => False
     end).
```

Figure 6.1: Type-Indexed Definition of Bisimilarity

polytypically: we can define a *type-indexed type* (Hinze et al., 2004). Using a type-indexed definition of bisimilarity $\approx\langle c\rangle$, we might state the property that map preserves identity as

$$\mathtt{Id}\langle\star\rangle\ T = \lambda f\ .\ \forall c : \mathtt{closed\_type}\ \ \star\ .\ \mathtt{decT}\ c = T \to \forall x : c\ .\ f\ x \approx\langle c\rangle\ x$$

In words: for all types T whose code in the universe is c and for all elements x of type c, $\mathtt{map}\langle c\rangle\ x$ is related to x in the bisimilarity relation specialized to c.

As it stands, our existing infrastructure for type-indexed *terms* can *almost* be used to define type-indexed types. Figure 6.1 gives the definition of type-indexed bisimilarity.

Unfortunately this definition is not accepted. In Section 2.4.1 we show that a polytypic type takes *np* type arguments and returns a term of type `decK star = Set`. In the definition of `Bisim`, the result type `A → A → Prop` lives in `Type`. To allow this definition we would need to change the decoder for kinds to return `Type` for kind $\star$ instead of `Set`. In Section 2.3.2 we explain our choice to use an impredicative sort as the decoding of kind $\star$, and although `Set` in Coq is (optionally) impredicative, `Type` is not. To change this from `Set` to `Type` we need to stratify our kind universe to eliminate the need for impredicativity.

One would hope that in a dependently typed language there is no essential difference between type-indexed values and type-indexed types. Type-indexed types are a crucial ingredient for some proofs (such as proofs about bisimilarity, as we have seen) and are useful to support other kinds of reasoning about polytypic functions. For instance, it should be possible to prove that every type in the universe denotes a container (see

Section 5.4.4), enabling semantic proofs about containers. However, this requires the computation of the type of shapes and positions: both type-indexed types. It would be interesting to see if a stratified kind universe introduces any unexpected difficulties and whether it indeed allows us to define type-indexed types.

6.4 Recursion

6.4.1 Satisfying the Guardedness Checker

In Chapter 4 we have seen that we can use the specialization of `map` to `list'`, which we call `mapList'`, to give a definition for `map` acting on `lists`:

```
Definition mapList' :=
 specTerm' list_struct map ((list, tt), ((list, tt), tt)).

CoFixpoint mapList A B f l :=
 toList (mapList' mapList f (fromList l)).
```

This works for two reasons:

1. The structural representation of `list'` has an occurrence of `list`, which it treats as a free variable. The specialization of `map` to `list'` therefore wants *two* functions: one for the elements of the list, and one for the tail of list—that is, `map` specialized to `list'` is a bifunctor. This setup can be compared to that in PolyP (Section 5.3.1) except that the variable denoting the recursion (`list` here and `rec` in the PolyP universe) is not given special treatment in our development.

2. After unfolding this definition, Coq will find that any recursive occurrence of `mapList` will be guarded by a `cons` constructor, inserted by `toList` (provided that the guardedness checker applies $\mu-$reduction, as explained in Section 4.3.2).

In principle, the same approach should work for proofs, and we demonstrated this in Section 4.4 using a handwritten proof for `list'`. Unfortunately, it fails when using proof specialization. The reason is that the proof given by proof specialization for `list'` is rather complicated—mostly due to the ubiquitous manipulation of heterogeneous equalities. The Coq guardedness checker is not sophisticated enough to unroll such a complicated definition to check for guardedness. In order to use proof specialization here, we need to simplify the resulting proofs.

In Section 5.4.2 we have discussed an alternative universe as defined by (Morris et al., 2006). One minor difference between their universe and ours is the inlining of free variable selection: their universe contains one constructor to select the first free variable and another to weaken the environment (throw one argument away). In their paper they say this makes proofs easier. It would be interesting to see if we can extend our universe in a similar manner, and what effect this has on the result of proof specialization.

However, even if such a universe simplifies proofs constructed by proof specialization, this change may not be enough. The proofs generated by proof specialization will have to be thoroughly examined to find out why they cause trouble during guardedness checking. We can then either attempt to simplify the generated proofs or to extend the guardedness checker further. This might be a process that requires patience, however: the proof that map preserves identity specialized to `list'` in normal form spans nearly 110 pages (*sic*).

6.4.2 Extending the Universe

The ideal solution to the problem of recursion would be an extension of the universe with support for recursion. This is not an easy problem, however. We have seen that in the related research on implementing generic programming in a dependent language (Section 5.4) only one such universe is given: the universe of strictly positive types. As explained, Morris et al. do not give a direct translation from a code in their universe to a datatype. Instead, they give a datatype which is *indexed* by a code: rather than giving the translation to a datatype they formalize what it means for a datatype corresponding to a code to be inhabited.

Although this is an ingenious idea, it does not easily scale to higher-order kinds. The decoding of a type variable of a kind other than $\star$ will be an uninhabited type. By directly formalizing what it means for a type corresponding to a code to be inhabited, such type variables cannot be treated. In particular, we cannot add type application to the universe in the same way—we cannot define what it means for a type corresponding to a code $F\ A$ to be inhabited by defining what it means for the types corresponding to F and A to be inhabited. This approach can therefore not be used for Generic Haskell-style (kind-indexed) generics. The definition of a Generic Haskell universe which includes recursion is a challenging research problem.

One possibility might be to take inspiration from some of the approaches mentioned in Section 5.5 which avoid syntactic checks for termination or guardedness. For instance, it might be possible to use the ordinal-based sized types approach from Abel (2009). It is not clear however if this approach can be ported to Coq, or whether we need a proof assistant that directly supports sized types (Abel (2009) mentions that he does not consider type checking sized types in the paper; Bertot and Komendantskaya (2009) cite the work by Abel and mention that it cannot currently be used in Coq). Moreover, Abel does not consider coinductive polytypic programs; he does consider coinduction in his thesis (Abel, 2006), which also gives a brief treatment of polytypic programming, but coinductive polytypic programs are mentioned as future work in (Abel, 2009).

6.5 Conclusions

Implementing polytypic programming (kind-indexed or otherwise) within a dependently typed language rather than as a language extension to or preprocessor for a functional language has significant benefits. Since we can formalize type specialization within the host language we get type checking of polytypic functions virtually for free. Further, polytypic function can be reified within the host language (that is, we can define a datatype that denotes polytypic functions) and are therefore first-class citizens: they can be passed around as arguments and returned as results. As a consequence, polytypic functions can be defined in terms of other polytypic functions and it becomes possible to define combinators on polytypic functions.

Throughout the development we needed to operate at the boundaries of type theory: we wrestled with universe inconsistencies and the absence of universe polymorphism, the need for impredicativity and ubiquitous reasoning about heterogeneous equalities. The syntactic checks for guardedness can be difficult to satisfy—when a proof is defined *this* way it is accepted, but when a proof is defined *that* way it is rejected. Semantic approaches to guardedness would be preferable.

Formalizing polytypic programs and proofs is therefore non-trivial, but once the infrastructure is in place that is no longer an issue. By reifying polytypic functions as a datatype `PolyFn`, Generic Haskell programmers will feel at home in Coq because they can define polytypic functions in a way that is familiar to them. The process of specialization is also known from Generic Haskell; the only difference is that specialization is now an ordinary (higher-order) function within the host language which takes a `PolyFn` as input and returns the specialized function as output which can then immediately be used in the definition of other functions.

Moreover, we have provided exactly the same infrastructure for proofs. By reifying polytypic proofs as a datatype `PolyProof`, the correspondence between polytypic functions and their polytypic proofs becomes very clear. Programmers need to give proofs for, and only for, the same cases that they need to give instances for when they define the polytypic function itself. Although we have to restrain the properties that can be expressed somewhat so that we can define proof specialization, these restrictions should not be too limiting in practise (especially given the relatively straight-forward extension we discuss in Section 6.3.1).

We set out to demonstrate that it is possible to do Generic Haskell-style polytypic programming within a dependently typed language, and that it is possible and straight-forward to do proofs over such programs; and this is what we have shown. *Qed.*

Appendix A

μ-Reduction

A copy of this file can also be found online (Verbruggen, 2009).

```
--- coq-8.2pl1-orig/kernel/inductive.ml
+++ coq-8.2pl1/kernel/inductive.ml
@@ -858,10 +858,92 @@
         with Not_found ->
      raise (CoFixGuardError (env, CodomainNotInductiveType b)))

+(*
+ * WV: New functions
+ * See thesis for explanation and justification.
+ *)
+
+(* Like zip in Haskell *)
+let array_zip arr1 arr2 =
+  assert (Array.length arr1 == Array.length arr2) ;
+  if (Array.length arr1 == 0)
+  then [| |]
+  else let result = Array.make (Array.length arr1)
+                      (Array.get arr1 0, Array.get arr2 0) in
+       for i = 0 to Array.length arr1 - 1 do
+         Array.set result i (Array.get arr1 i, Array.get arr2 i)
+       done ;
+       result ;;
+
+(* Like unzip in Haskell *)
+let array_unzip arr =
+  if Array.length arr == 0
+  then ([| |], [| |])
+  else let (x, y) = Array.get arr 0 in
+       let arr1 = Array.make (Array.length arr) x in
+       let arr2 = Array.make (Array.length arr) y in
+       for i = 0 to Array.length arr - 1 do
+         let (x, y) = Array.get arr i in
+         Array.set arr1 i x ;
+         Array.set arr2 i y
+       done ;
+       (arr1, arr2) ;;
+
+(* f o \x. \y. e |-> \x. \y. f e *)
+let compose_constr env f g n =
+  let rec compose_constr_rec f g n =
+    if n == 0
+    then (mkApp (f, [| g |]))
+    else match kind_of_term g with
+         | Lambda (x, t1, t2) ->
+            mkLambda (x, t1, compose_constr_rec (lift 1 f) t2 (n - 1))
+         | _                  -> assert false
+  in compose_constr_rec f g n ;;
+
+(* Find (strong) beta-iota-delta-zeta-mu normal form *)
+let betadeltaiotamu env e =
+  let rec mu_rec e =
+    let e' = Reduction.whd_betadeltaiota env e in
+    match kind_of_term e' with
+    | App (f, args) ->
+       let (f', ch_f) = mu_rec f in
+       let (args', ch_args) = array_unzip (Array.map mu_rec args) in
+       (mkApp (f', args'), ch_f || Array.fold_right (||) ch_args false)
+    | Case (ci, p, t, br) ->
+        let (p', ch_p) = mu_rec p in
+        let (t', ch_t) = mu_rec t in
+        let (br', ch_br) = array_unzip (Array.map mu_rec br) in
+        let def = (mkCase (ci, p', t', br'),
+                   ch_p || ch_t || Array.fold_right (||) ch_br false) in
+        (match kind_of_term t with
+        | Case (ci',p',t',br') ->
+            let argtype =
+              (match kind_of_term p with
+               | Lambda (x, t1, t2) -> t1
+               | _ -> assert false) in
+            let fn = mkLambda (Anonymous, argtype,
+                               (mkCase (ci, lift 1 p,
+                                        mkRel 1, Array.map (lift 1) br))) in
+            let new_branches =
+             Array.map (fun (br, n_args) -> compose_constr env fn br n_args)
+                       (array_zip br' ci'.ci_cstr_nargs) in
+            (mkCase (ci', compose_constr env p p' 1, t', new_branches), true)
+        | _ -> def)
+    | _  -> (e', false)
+  and repeat_mu_rec e =
+    let (e', ch_e) = mu_rec e in
+    if ch_e then repeat_mu_rec e' else e'
+  in repeat_mu_rec e
+
+(*
+ * WV: end new functions
+ *)
+
 let check_one_cofix env nbfix def deftype =
   let rec check_rec_call env alreadygrd n vlra  t =
     if not (noccur_with_meta n nbfix t) then
-      let c,args = decompose_app (whd_betadeltaiota env t) in
+      (* WV: call betadeltaiotamu instead of whd_betadeltaiota *)
+      let c,args = decompose_app (betadeltaiotamu env t) in
      match kind_of_term c with
   | Rel p when  n <= p && p < n+nbfix ->
       (* recursive call: must be guarded and no nested recursive
```

Bibliography

Andreas Abel. *Type-Based Termination: A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Fakültat für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München, 2006.

Andreas Abel. Type-based termination of generic programs. *Science of Computer Programming*, 74(8):550–567, 2009. Special Issue on Mathematics of Program Construction (MPC'06).

Michael Abott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In *Proceedings of Foundations of Software Science and Computation Structures*, 2003.

Artem Alimarine. *Generic Functional Programming: Conceptual Design, Implementation and Applications*. PhD thesis, Radboud Universiteit Nijmegen, The Netherlands, 2005.

Artem Alimarine and Rinus Plasmeijer. A generic programming extension for clean. In *IFL'01: Proceedings of the 13th International Workshop on the Implementation of Functional Languages*, volume 2312 of *Lecture Notes in Computer Science*, pages 168–185. Springer–Verlag, 2001.

Artem Alimarine and Sjaak Smetsers. Improved fusion for optimizing generics. In *PADL'05: Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, volume 3350 of *Lecture Notes in Computer Science*, pages 203–218. Springer–Verlag, 2005.

Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In *MPC'04: Proceedings of the 7th International Conference on Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 16–31. Springer–Verlag, 2004.

Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 1–20. Kluwer, B.V., 2003.

Thorsten Altenkirch, Conor Mcbride, and Peter Morris. Generic programming with dependent types. In *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer–Verlag, 2007.

Roland Backhouse and Paul Hoogendijk. Generic properties of datatypes. In *Generic Programming: Advanced Lectures*, volume 2793 of *Lecture Notes in Computer Science*, pages 97–132. Springer–Verlag, 2003.

Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming—an introduction. In *AFP'98: Advanced Functional Programming, Third International School*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–108. Springer–Verlag, 1998.

Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

Yves Bertot. Filters on coinductive streams, an application to eratosthenes' sieve. In *TLCA'05: Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications*, volume 3461 of *Lecture Notes in Computer Science*, pages 102–115. Springer–Verlag, 2005.

Yves Bertot and Pierre Castéran. *Coq'Art: Interactive Theorem Proving and Program Development*. Springer–Verlag, 2004.

Yves Bertot and Ekaterina Komendantskaya. Inductive and coinductive components of corecursive functions in Coq. *Electronic Notes in Theoretical Computer Science*, 203 (5):25–47, 2008.

Yves Bertot and Ekaterina Komendantskaya. Using structural recursion for corecursion. In *Types for Proofs and Programs*, volume 5497 of *Lecture Notes in Computer Science*, pages 220–236. Springer–Verlag, 2009.

Ana Bove. *General Recursion in Type Theory*. PhD thesis, Department of Computing Science, Chalmers University of Technology, November 2002.

Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18, 2005.

Timothy Colburn and Gary Shute. Abstraction in computer science. *Minds and Machines*, 17(2):169–184, July 2007.

Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.

Nils Anders Danielsson and Thorsten Altenkirch. Mixing induction and coinduction. Draft, available online, 2009.

N. G. de Bruijn. A lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–392, 1972.

Maarten de Mol, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. Theorem proving for functional programmers - Sparkle: A functional theorem prover. In *IFL'02: Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, pages 55–71. Springer–Verlag, 2002.

Maarten M. Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, Centre of Mathematics and Computer Science, CWI, Amsterdam, Jan 1991.

Pietro Di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In *Types for Proofs and Programs*, volume 2646 of *Lecture Notes in Computer Science*, pages 148–161. Springer–Verlag, 2003.

Jeremy Gibbons. Datatype-generic programming. In *School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 1–71. Springer–Verlag, 2006.

Jeremy Gibbons and Ross Paterson. Parametric datatype-genericity. In *WGP'09: Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, pages 85–93. ACM Press, 2009.

Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

Ralf Hinze. Polytypic values possess polykinded types. In *MPC'00: 5th International Conference on Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer–Verlag, 2000a.

Ralf Hinze. *Generic Programs and Proofs*. Habilitationsschrift, Universität Bonn, Germany, 2000b.

Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory. In *Generic Programming: Advanced Lectures*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer–Verlag, 2003.

Ralf Hinze and Andres Löh. Generic programming, now! In *School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 150–208. Springer–Verlag, 2006a.

Ralf Hinze and Andres Löh. "scrap your boilerplate" revolutions. In *MPC'06: Mathematics of Program Construction: 8th International Conference*, volume 4014 of *Lecture Notes in Computer Science*, pages 180–208. Springer–Verlag, 2006b.

Ralf Hinze and Andres Löh. Generic programming in 3D. *Science of Computer Programming*, 74:590–628, 2009.

Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming, MPC Special Issue*, 51(1-2):117–151, 2004.

Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in Haskell. In *School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 72–149. Springer–Verlag, 2006a.

Ralf Hinze, Andres Löh, and Bruno Oliveira. "scrap your boilerplate" reloaded. In *Proceedings of FLOPS 2006*, 2006b.

Paul Hoogendijk and Oege de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, 2000.

Antonius J. C. Hurkens. A simplification of girard's paradox. In *TLCA'95: Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, pages 266–278. Springer–Verlag, 1995.

P. Jansson and J. Jeuring. PolyP—a polytypic programming language extension. In *POPL'97: Conference Record 24th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

C. Barry Jay. A semantics for shape. In *ESOP'94: Selected papers of ESOP '94, the 5th European symposium on Programming*, pages 251–283. Elsevier Science Publishers B. V., 1995.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *TLDI'03: ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, volume 38, pages 26–37. ACM Press, 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: Extensible generic functions. In *ICFP'05: Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, pages 204–215. ACM Press, 2005.

Andres Löh. *Exploring Generic Haskell*. PhD thesis, Instituut voor Programmatuurkunde en Algoritmiek, Utrecht, The Netherlands, 2004.

Zhaohui Luo and Randy Pollack. Lego proof development system: User's manual. Technical Report ECS-LFCS-92-211, Edinburgh University, 1992.

Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

Conor McBride. Elimination with a motive. In *TYPES'00: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 197–216. Springer–Verlag, 2002.

Peter Morris and Thorsten Altenkirch. Indexed containers. In *Twenty-Fourth IEEE Symposium in Logic in Computer Science (LICS 2009)*, 2009. To appear.

Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In *TYPES'04: Types for Proofs and Programs*, volume 3839 of *Lecture Notes in Computer Science*, pages 252–267. Springer–Verlag, 2006.

Peter Morris, Thorsten Altenkirch, and Neil Ghani. A universe of strictly positive families. *Theory of Computation*, 2007a.

Peter Morris, Thorsten Altenkirch, and Neil Ghani. Constructing strictly positive families. In *CATS'07: Theory of Computing*, volume 65 of *Conferences in Research and Practice in Information Technology (CRPIT)*, 2007b.

Milad Niqui. Coalgebraic reasoning in Coq: Bisimulation and the λ-coiteration scheme. In *TYPES'08: Types for Proofs and Programs, International Conference, Torino, Italy, March 26–29, 2008. Revised Selected Papers*, volume 5497 of *Lecture Notes in Computer Science*, pages 272–288. Springer–Verlag, 2009.

Ulf Norell. Functional generic programming and type theory. Master's thesis, Computing Science, Chalmers University of Technology, 2002.

Sam Owre, Sreeranga P. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining specifications, proof checking and model checking. In *CAV'96: Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer–Verlag, 1996.

Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

Simon Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *ESOP'96: Proceedings of the European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*. Springer–Verlag, 1996.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, 2006.

Holger Pfeifer and Harald Rueß. Polytypic proof construction. In *TPHOLs'99: Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pages 55–72. Springer–Verlag, 1999.

Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: Executable specifications of interactive work flow systems for the web. In *ICFP'07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 141–152. ACM Press, 2007.

Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

F. Reig. Generic proofs for combinator-based generic programs. *Trends in Functional Programming*, 5, 2006.

Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell'08: Proceedings of the First ACM SIGPLAN Symposium on Haskell*, pages 111–122. ACM Press, 2008.

Alexey Rodriguez, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP'09: Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 233–244. ACM Press, 2009.

Tim Sheard. Generic programming in ωmega. In *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer–Verlag, 2007.

M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006.

The Coq Development Team. Coq frequently asked questions (v8.1), 2008a. `http://www.lix.polytechnique.fr/coq/node/16`.

The Coq Development Team. Coq reference manual (version 8.2), 2008b. `http://www.lix.polytechnique.fr/coq/refman/`.

Wendy Verbruggen. Coq source files, 2009. `http://www.cs.tcd.ie/~verbruwj/src/`.

Wendy Verbruggen, Edsko de Vries, and Arthur Hughes. Polytypic programming in Coq. In *WGP'08: Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, pages 49–60. ACM Press, 2008.

Wendy Verbruggen, Edsko de Vries, and Arthur Hughes. Polytypic properties and proofs in Coq. In *WGP'09: Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, pages 1–12. ACM Press, 2009.

Wendy Verbruggen, Edsko de Vries, and Arthur Hughes. Formal polytypic programs and proofs. *Journal of Functional Programming*, 2010. To appear.