Under consideration for publication in J. Functional Programming

Formal Polytypic Programs and Proofs

WENDY VERBRUGGEN∗, EDSKO DE VRIES† and ARTHUR HUGHES
School of Computer Science and Statistics
Trinity College Dublin, College Green, Ireland
(*e-mail:* wendyverbruggen@gmail.com, {Edsko.de.Vries,Arthur.Hughes}@scss.tcd.ie)

Abstract

The aim of our work is to be able to do fully formal, machine verified proofs over Generic Haskell-style polytypic programs. In order to achieve this goal, we embed polytypic programming in the proof assistant *Coq* and provide an infrastructure for polytypic proofs. Polytypic functions are reified within Coq as a datatype and they can then be specialized by applying a dependently typed term specialization function. Polytypic functions are thus first class citizens and can be passed as arguments or returned as results. Likewise, we reify polytypic proofs as a datatype, and provide a lemma that a polytypic proof can be specialized to any datatype in the universe. The correspondence between polytypic functions and their polytypic proofs is 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. Finally, we discuss how to write (co)recursive functions and do (co)recursive proofs in a similar way that recursion is handled in Generic Haskell.

1 Introduction

In the never ending quest for higher levels of abstraction in programming language research, *generic programming* has been a popular research topic in the functional programming community for a while (Jansson & Jeuring, 1997; Hinze & Peyton Jones, 2001; Lämmel & Visser, 2002; Lämmel & Peyton Jones, 2003; Hinze, 2006; Hinze & Löh, 2006; Hinze & Löh, 2009; Rodriguez *et al.*, 2009). Unfortunately, a consensus on the best approach has yet to be reached, and the number of approaches to generic programming almost equals the number of papers written on the topic. The subject area can be bewildering; some survey papers by Hinze *et al.* (2006) and Rodriguez *et al.* (2008) try to disentangle some of the various strands of research.

One particular strand that we are interested in is *polytypic programming* as advocated by Hinze in his seminal habilitationsschrift (2000), which has been incorporated in at least two language designs: Generic Haskell (Löh, 2004) and Generic Clean (Alimarine, 2005). The polytypic programming style of Generic Haskell is characterized by the use of kind-indexed types. The key 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

∗ Supported by the Irish Research Council for Science, Engineering and Technology.
† This research was supported by SFI project SFI 06 IN.1 1898.

$F\langle T\rangle$ 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.

We extend this paradigm to proofs over polytypic functions. Like polytypic types, polytypic properties are kind-indexed and like polytypic functions, polytypic proofs are type-indexed. The aim of this paper is to provide an infrastructure within the proof assistant Coq that makes it possible to do formal (in the sense of "machine verified") proofs over Generic Haskell-style polytypic functions. We make the following contributions:

1. 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.
2. In Generic Haskell individual instances of specialization are type checked by the Haskell compiler but the polytypic function itself is not type checked. In contrast, we only allow the user to define type correct polytypic functions and we formally prove that the result of term specialization (`specTerm T f`) must have the type computed by type specialization (`specType T F`).
3. We give a definition of a polytypic property and show how it can be specialized. It is not always easy to formulate polytypic properties, and the fact that we may now ask the proof assistant to specialize polytypic properties to specific datatypes can be helpful when trying out ideas.
4. We provide an infrastructure for *polytypic proofs* similar to the infrastructure for polytypic functions, so that formal proofs over polytypic functions can be done with little effort.
5. We give a number of example polytypic properties and give corresponding polytypic proofs.
6. The infrastructure for polytypic proofs can be seen as a formal proof that
   - property specialization yields well-formed properties, and that
   - proof specialization is correct with respect to property specialization.
7. As we will see, when the user gives a polytypic function or a polytypic proof, he only needs to provide instances of the function or of the proof for the type constants in the universe. Our development is a formal proof that such a function definition can indeed be specialized to *any* type in the universe, and that such a proof indeed suffices to prove the property for *any* type in the universe.
8. We discuss how to define corecursive functions and do coinductive proofs over polytypic functions. Unfortunately, recursion 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.

Figure 1 gives a bird's eye view of the paper. The Coq sources associated with the formalization described in this paper can be found online (Verbruggen, 2009).

Since this is a long paper, most sections will start with a "roadmap" which will explain what has been achieved so far, what the subject is of the new section, and how the section relates to what is yet to come. We hope that this will aid readability and avoid the reader getting lost. Here we will give a bird's eye view of the paper and introduce our running example, polytypic map.

Section 2 is an introduction to Coq. We assume the reader is familiar with Generic Haskell but may not be comfortable with Coq. The aim of this section is to introduce the most important concepts.

Sections 3 and 4 explain how we define polytypic functions in a style that should be familiar from Generic Haskell. Section 4.1 will introduce polytypic *types* and give the type of polytypic map:

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

Section 4.2 introduces polytypic *functions* and defines polytypic map:

```
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).
```

Sections 5 and 6 describe how polytypic types and terms are specialized to specific datatypes.

Section 7 turns to proofs of polytypic *properties*. We will show how to state the functor laws for map polytypically; for example, preservation of composition is given in Section 7.2 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).
```

Section 7.3 explains how to do polytypic *proofs* and gives the notably concise proof that map satisfies `Comp`:

```
Lemma map_Comp : PolyProof map Comp.
Proof.
  apply (polyProof map Comp); compute; auto; intros.
  destruct x; rewrite H; rewrite H0; auto.
  destruct x; [rewrite H | rewrite H0]; auto.
Defined.
```

Sections 8 and 9 are the counterparts of Section 5 and Section 6 and explain property and proof specialization. As an example, the specialization of the polytypic property `Comp` of preservation of composition to the datatype `fork` $(\Lambda \mathtt{A} \,.\, \mathtt{A} \times \mathtt{A})$ of kind $\star \to \star$ gives:

```
∀(A B C : Set) (f : B → C) (g : A → B) (h : A → C),
  (∀x : A, f (g x) = h x) → ∀(x,y) : A × A, (f (g x),f (g y)) = (h x,h y)
```

Section 10 discusses how to deal with recursive definitions. Since Haskell datatypes capture both finite and infinite data structures, we will focus on corecursion.

Sections 11 and 12 finally discuss related and future work, and we will conclude in Section 13.

Fig. 1. Bird's eye view of the paper

2 Coq

Before we delve into our formalization of polytypic programming, we will give a brief overview of Coq. We discuss induction and recursion in Section 2.1, implicit arguments in Section 2.2, coinduction and corecursion in Section 2.3, and dependent types and their use in proofs in Sections 2.4 and 2.5. Finally, we discuss universes in Section 2.6 and the treatment of equality in Section 2.7. Readers familiar with Coq can skip this section, although they may wish to glance over our definition of heterogeneous tuples in Section 2.6 and our definition of `convert` in Section 2.7 as we will be using those in the rest of the development.

Coq is a proof assistant developed in INRIA based on the calculus of constructions, i.e. higher-order predicate logic, extended with inductive and coinductive datatypes and an infinite hierarchy of universes. We can give but a brief overview of Coq here. For more information, we refer the reader to the excellent textbook on Coq (Bertot & Castéran, 2004).

2.1 Induction and Recursion

Inductive datatypes are introduced in much the same way that algebraic datatypes are introduced in Haskell, using a syntax reminiscent of GADTs (Schrijvers *et al.*, 2009). We will illustrate this with some examples. The simplest type we can define is the empty type:

```
Inductive Empty_set : Set :=.
```

"`Set`" denotes the type of the type itself and corresponds to kind $\star$ in Haskell. Only slightly more interesting is the unit type, which is denoted by `()` in Haskell, which is called `unit` in Coq; its sole inhabitant is called `tt`. The type is defined as

```
Inductive unit : Set :=
 | tt : unit.
```

The type of pairs, `(,)` in Haskell, is parameterized by types `A`, `B` and can be defined in Coq as

```
Inductive prod (A : Set) (B : Set) : Set :=
 | pair : A → B → prod A B.
```

Similarly, we can construct the sum of two types, denoted by `Either` in Haskell, as

```
Inductive sum (A : Set) (B : Set) : Set :=
 | inl : A → sum A B
 | inr : B → sum A B.
```

We will also make use of Coq's `option` type, which corresponds to `Maybe` in Haskell:

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

Inductive datatypes also give us recursion at the type level. For example, we can construct the type of natural numbers as follows:

```
Inductive nat : Set :=
 | O : nat
 | S : nat → nat.
```

This is the Peano encoding of the natural numbers. Finally, we can define the type of lists of elements of a type `A` as

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

Functions over these inductive datatypes are defined much in the same way as their Haskell counterparts. For example, here is the standard map function over lists:

```
Fixpoint map (A B : Set) (f : A → B) (xs : list A) : list B :=
 match xs with
  | nil ⇒ nil B
  | cons x xs' ⇒ cons B (f x) (map A B f xs')
 end.
```

The `match` construct corresponds to `case` analysis in Haskell. Unlike in Haskell, however, all recursive functions must be total, i.e. terminate, in Coq and this is enforced by a syntactic restriction: at least one of the arguments to the recursive call must be getting structurally smaller. In the case of `map` the list argument `xs` decreases in length in each recursive call.

Another thing to note about this definition is the explicit use of the type arguments `A` and `B` for `nil` and `cons` on the right-hand side, whereas they are absent on the left-hand side. Generally, all uniform parameters—parameters that do not differ from a term to its subterms—are not mentioned when pattern matching on a term, but must be given when *constructing* a term. See Section 6.5.2, *Variably Dependent Inductive Types*, of Coq'Art for details (Bertot & Castéran, 2004).

Coq introduces some syntactic sugar for the types defined above: ordinary numbers can be used to give instances of `nat`, `A × B` corresponds to `prod A B`, `(x, y)` corresponds to `pair x y` and `A + B` corresponds to `sum A B`. Unlike Haskell, Coq uses a different syntax for values and their types: `(x, y)` is a pair, whereas `A × B` is the type of a pair. This is important because `(A, B)` in Coq is a pair containing two types, and has type `Set × Set`.

2.2 Implicit Arguments

Type arguments are explicit arguments in Coq, which is why we pass `B` as an argument to `nil` and `cons` and both `A` and `B` as arguments to the recursive call to `map`. However, in many cases Coq can actually infer arguments automatically. We can therefore also write `map` as

```
Fixpoint map (A B : Set) (f : A → B) (xs : list A) : list B :=
 match xs with
  | nil ⇒ nil _
```

```
  | cons x xs' ⇒ cons _ (f x) (map _ _ f xs')
 end.
```

The underscores indicate arguments that we are asking Coq to infer for us. Moreover, we can declare some arguments to be implicit by default. For example, we can declare the type arguments of `nil` and `cons` to be implicit:

```
Implicit Arguments nil [A].
Implicit Arguments cons [A].
```

We can even ask Coq to automatically make as many arguments implicit as possible:

```
Set Implicit Arguments.
```

With these declarations in place, we can write `map` more concisely as

```
Fixpoint map (A B : Set) (f : A → B) (xs : list A) : list B :=
 match xs with
  | nil ⇒ nil
  | cons x xs' ⇒ cons (f x) (map f xs')
 end.
```

2.3 Coinduction and Corecursion

Unlike Haskell's datatypes, inductive datatypes in Coq only have finite inhabitants: the list datatype above only describes finite lists (Bertot & Castéran, 2004). Types inhabited by infinite terms are described by coinductive datatypes. For instance, here is a definition of *streams*, describing infinite lists:

```
CoInductive stream (A : Set) : Set :=
 | scons : A → stream A → stream A.
```

Corecursive functions over coinductive datatypes are defined in much the same way as recursive functions, but unlike recursive functions they do not need to terminate. However, they must be *productive*. Intuitively, they must always generate the next part of the output in finite time. This is enforced syntactically by requiring that every recursive call must be an argument to a constructor of the datatype. We will give a more precise definition of corecursion in Section 10. An easy example of a guarded corecursive function is map for streams:

```
CoFixpoint smap (A B : Set) (f : A → B) (xs : stream A)
  : stream B :=
 match xs with
  | scons x xs' ⇒ scons (f x) (smap f xs')
 end.
```

The standard example of a function which is *not* productive is `filter`, which returns the elements of a list that satisfy a predicate. We might attempt to define it like

```
CoFixpoint sfilter (A : Set) (p : A → bool) (xs : stream A)
```

```
 : stream A :=
match xs with
 | scons x xs' ⇒ if p x then scons x (sfilter p xs')
                        else sfilter p xs' (* Not guarded! *)
end.
```

This definition is rejected because the second call to `sfilter` is not guarded. It is not difficult to see that `filter` is not productive: it may be that none of the elements in the stream satisfy the predicate so that `sfilter` never produces any part of the result. Indeed, we can only define a filter on streams if we are given a proof that there is always a next element of the stream that satisfies the predicate (Bertot & Komendantskaya, 2008).

2.4 Dependent Types

The real power of Coq and the major difference with Haskell comes from the fact that Coq features dependent types: types are first class and can be passed as arguments to functions or computed as results.

For example, suppose that we want to describe a type of homogeneous tuples of length `n` with values of type `A` (sometimes called *vectors*). We might describe this type as $(\mathtt{A},\ldots,\mathtt{A})$ in Haskell but we have no way to define it. But we can easily construct this type in Coq:

```
Fixpoint tupleS (A : Set) (n : nat) : Set :=
 match n with
 | O ⇒ unit
 | S m ⇒ A × tupleS A m
 end.
```

For the zero-tuple we return the unit type, and for the tuple of length at $\mathtt{m}+1$ we return the type of pairs of an element of type `A` and a tuple of length `m`. Here is an example of a tuple of length 3 containing natural numbers:

```
Definition exampleNatTuple : tupleS nat 3 := (8, (3, (42, tt))).
```

To define a function that returns the *i*th element from an *n*-tuple, we must first define a datatype that describes the set of valid indices into the tuple:

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

For example, we have that

the expression	*evaluates to the type*	*which comprises*
`index 0`	`Empty_set`	{}
`index 1`	`option Empty_set`	{None}
`index 2`	`option (option Empty_set)`	{None, Some None}

In words, there are no valid indices into an empty tuple, there is only a single index into a singleton tuple, etc. Using this index type, we can write the function that gets the *i*th element from a tuple as:

```
Fixpoint getS (A : Set) (n : nat) : index n → tupleS A n → A :=
 match n return index n → tupleS A n → A with
 | O ⇒ fun i _ ⇒ match i with end
 | S n' ⇒ fun i tup ⇒
     match i with
     | None ⇒ fst tup
     | Some i' ⇒ getS A n' i' (snd tup)
     end
 end.
Implicit Arguments getS [A n]
```

The "`match` n `return` τ `with`" defines a pattern match where the type of each branch τ may depend on n, similar to GADTs in Haskell. This is necessary in `getS`. For example, when the pattern match finds that we are dealing with the empty tuple $(n = 0)$, we need to know that the type of i is `index 0`. The pattern match on i will then have no branches because the empty set has no constructors, and we are done immediately. For the case where the tuple has length at least one, we check the index to see if we need to return the first element of the tuple or whether we need to recurse.

Throughout this paper we will often replace the syntax of indices by their corresponding natural numbers for readability, writing "0" for `None`, "1" for `Some None`, "2" for `Some (Some None)`, etc.

2.5 Proofs

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

Coq introduces a universe `Prop` for propositions, at the same level as `Set`. `Prop` is used to separate computational content from proofs, which is necessary to support extraction of Coq code to languages such as Haskell. If A : `Prop` is a proposition then to prove A it suffices to give a term $a : A$. Totality is essential here: "non-terminating" proofs are not really proofs at all.

The basic datatypes we have seen in `Set` have counterparts in `Prop`:

- The empty datatype corresponds to the proposition `False`, which has no proofs.
- The unit datatype corresponds to the proposition `True`, which is trivially proven.
- The pair datatype corresponds to the logical conjunction of two propositions; its elements are pairs of proofs of both propositions.
- The sum datatype corresponds to the logical disjunction of two propositions; its elements are proofs of one of the two propositions.

As a more interesting example, consider the inductive type that corresponds to the proposition that a natural number `n` is even:

```
Inductive even : nat → Prop :=
 | even_0 : even 0
 | even_SS : ∀ (n : nat), even n → even (S (S n)).
```

This is a dependent datatype, as it depends on a value, a natural number. Here is a proof that 4 is even:

```
Lemma even_4 : even 4 := even_SS (even_SS even_0).
```

`Lemma` is just syntactic sugar for `Definition` to make intent clearer; there is no distinction as far as Coq is concerned. As another simple example, consider 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).
```

Given two propositions `A` and `B`, a proof `a` of `A` and a proof `f` that `A` implies `B`, `MP` constructs a proof of proposition `B` simply by applying `f` to `a`.

For more complicated proofs, we may choose to make use of *tactics*. Tactics are small programs that search for proofs in a particular domain. The use of tactics enables *proof automation*, where Coq handles 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* for writing custom tactics. Since tactics return a proof if one can be found, this proof can be verified so that "rogue" tactic cannot compromise the soundness of the system. A deeper understanding of tactics will not be required to read this paper, so we refer the reader to Coq'Art (Bertot & Castéran, 2004) for more information. However, the support for tactics and proof automation is an important reason for choosing Coq for our work since we feel that they will ease the burden on users of our system.

2.6 Universes

We have seen that "5" has type `nat` in Coq, that `nat` has type `Set`, and that `Set` corresponds to kind $\star$ in Haskell. 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 $\texttt{T} : \texttt{Set}$ then $\texttt{T} : \texttt{Type}_0$ and if $\texttt{T} : \texttt{Type}_i$ then $\texttt{T} : \texttt{Type}_j$ for any $j \geq i$. This *stratification* of `Type` prevents the encoding of logical paradoxes (Hurkens, 1995). Universe levels are not written explicitly in Coq code, where we simply write "`Type`", but are inferred by the type checker.

For example, a generalization of `tupleS` from the previous section is

```
Fixpoint tupleT (A : Type) (n : nat) : Type := ..
```

where `A` has type `Type` rather than `Set`. With this new definition, we can create a tuple of types of type `Set`:

```
Definition exampleSetTuple : tupleT Set 2 := (nat, (unit, tt)).
```

One definition that we will need later in our proofs 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 $A \times B \times C$:

```
gtupleT : ∀ n : nat,  tupleT Type n → Type
```

Such a function 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

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

we will run into a `Universe inconsistency` error.

To understand this error, we need to see the universe constraints inferred by the type checker. For `tupleT` we get

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

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

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:

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

Since we pass $\texttt{Type}_j$ as the first argument to `tupleT`, and we have said that the first argument has type $\texttt{Type}_i$, we must have $\texttt{Type}_j : \texttt{Type}_i$. This constraint 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 & Pollack, 1991). A work-around would be to duplicate the definition of `tupleT` which would then have type $\texttt{Type}_{i'} \to \texttt{nat} \to \texttt{Type}_{j'}$. This duplicate definition of `tupleT` 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.* (2007) 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 $(\texttt{x},\texttt{y},\texttt{z})$ of elements of some type `A` and a function $\texttt{f} : \texttt{A} \to \texttt{Type}$, we construct the type $\texttt{f x} \times \texttt{f y} \times \texttt{f z}$. This alternative definition is implemented as

```
Fixpoint gtupleT (A : Set) (n : nat) (f : A → Type)
  : tupleS A n → Type :=
 match n return tupleS A n → Type with
 | 0   ⇒ fun _ ⇒ unit
 | S m ⇒ fun tup ⇒ f (fst tup) × gtupleT m f (snd tup)
 end.
Implicit Arguments gtupleT [A n].
```

While this definition of `gtupleT` is not formally equivalent to the previous one, it is equally suitable for our purposes and avoids the universe inconsistency by avoiding feeding the result of `tupleT` back into `tupleT`.

2.7 Reasoning about Equality

The standard definition of equality in Coq states that two terms of the same type which reduce to the same normal form are equal:

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

This 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) generalizes the standard equality relation and allows us to state equalities between terms of different types, even though its only constructor still only allows us to prove equality between terms of the same type:

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

To prove $(e_1 : T_1) \simeq_{T_1,T_2} (e_2 : T_2)$ we must first show that $T_1 = T_2$ and 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

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

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

When terms get larger it is not always obvious what we need to generalize over and in which order. Moreover, suppose we have some dependent type $D : T \to \texttt{Set}$, a function $f : \forall (t : T), D\ t \to T'$, two elements $t_1, t_2 : T$ and $d_1 : D\ t_1$, $d_2 : D\ t_2$, and that we know that $d_1 \simeq_{D\ t_1, D\ t_2} d_2$ (but $t_1 \neq t_2$). 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 equality may or may not be difficult to prove.

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

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

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 and B are implicit. Associated with `convert` is a lemma proving that this conversion does not change the actual element, only its type:

Lemma 1 (*Convert Identity*)

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

However, even armed with Lemma 1, proofs about heterogeneous equality are not straightforward as `convert` $p\ x$ cannot simply be replaced by x since this would yield ill-formed terms. For example, proving that

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

may be difficult: it needs to be proven as a property of f, but if f is defined by structural induction on its second argument 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 to facilitate induction. For example, if d_1 is a list, convert each element of the list rather than converting the list itself.

3 Definition of the Generic View

In Generic Haskell, generic functions are defined by induction on the structure of datatypes. While types are first-class in Coq, we cannot inspect or discriminate them. Instead we define an inductive datatype, a type of codes, whose elements are interpreted as types. This allows us to define specialization of generic functions by induction on the structure of these codes.

For example, `tprod` is a code in the universe that corresponds to the Coq product type. If we want to specialize the polytypic map function to products, we pass `tprod` as an argument to term specialization, and term specialization is defined by induction on the structure of this argument.

However, the result of term specialization, should be a function on the actual Coq datatype for products. To accomplish this we need a mapping from codes to Coq datatypes, such a mapping is known as a *decoder*.

Similarly we need to encode kinds and provide the associated decoder. The codes for types and kinds are often called a *generic view* or a *universe*. The definitions for our generic view are given in Figure 2, and the decoders for kinds and types are defined in Figure 3. Our universe for types encodes only well-kinded types; this is explained in Section 3.1. The decoders for kinds and types are discussed in Sections 3.2 and 3.3, and we conclude with a few examples in Section 3.4.

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

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

(* Kind environment for free variables *)
Definition envk (nv : nat) : Set := tupleS kind nv.

(* 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).
Implicit Arguments type [].
Implicit Arguments tconst [nv ek k].
Implicit Arguments tvar [].
Implicit Arguments tapp [nv ek k1 k2].
Implicit Arguments tlam [nv ek 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
```

Fig. 2. Generic View

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 (see Figure 2). As an example, the rule `tlam` for lambda abstraction encodes the kinding derivation

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

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

(* Decoder for types *)
Fixpoint decT (nv : nat) (k : kind) (ek : envk nv) (t : type nv ek k)
  : 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      (* Coq unit type *)
              | tc_int ⇒ Z          (* Coq type of integers *)
              | tc_prod ⇒ prod_set (* Coq product type in Set *)
              | tc_sum ⇒ sum_set    (* Coq sum type in 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.
Implicit Arguments decT [nv k ek].
```

Fig. 3. Decoders

We use De Bruijn indices to represent variables (de Bruijn, 1972). The indices in a type of *nv* free variables cannot be out of bounds since their type is `index` *nv* (Section 2.4).

3.2 Decoding Kinds

The decoder `decK` for kinds is straight-forward except for a subtlety in the choice of `Set` for kind $\star$. During type specialization (Section 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 2.6, `Type` in Coq is not impredicative so that using `Type` for (`decK star`) will result in a universe inconsistency. Hence we choose `Set` instead, enabling the impredicative `Set` option[1].

Another way to solve the impredicativity 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

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

[1] Non-impredicative `Set` is useful mostly for classical reasoning, which we make no use of. Making `Set` impredicative therefore does not compromise soundness (Coq Development Team, 2008a; Coq Development Team, 2008b). Impredicative `Set` might also be justified since we are interested in doing proofs over Generic Haskell-style programs, and Haskell supports impredicative types (Vytiniotis *et al.*, 2006).

Here it is possible to 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 used stratified kinds.

3.3 Decoding Types

The decoder `decT` 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 $\mathtt{T}_i$ with every free variable i in `T`. Since the kind of $\mathtt{T}_i$ depends on the kind 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 `gtupleT` described in Section 2.6.

Armed with the type environment `et` we define the decoder for types `decT` as shown in Figure 3. 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.

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. We have added some notational shorthand to make these example types more readable:

```
Notation "t @ s" := (tapp t s) (at level 30).
Notation "t + s" := (tapp (tapp (tconst tc_sum) t) s).
Notation "t × s" := (tapp (tapp (tconst tc_prod) t) s).
Notation "1"     := (tconst tc_unit).
```

Unfortunately the definitions are still a little heavy on notation, especially in dealing with type variables—`var` i represents the ith variable—but additional syntactic sugar is left to future work.

Consider the type `fork`, defined in Haskell as

```
data Fork a = MkFork a a
```

We encode this as a type with one argument $\Lambda A\,.\,A \times A$ 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 $\star \to \star$. We decode `fork` to a Coq type using our type decoder `decT`, where `tt` represents the empty type environment:

```
Eval compute in decT fork tt.
 = (fun A : Set ⇒ A × A) : decK (karr star star)
```

The command `Eval r in x` performs the reductions specified by r on the term x and displays the resulting term with its type. In this case we use the tactic *compute* to specify the reductions, which represents call-by-value β, δ, ι and ζ-reduction.

We can evaluate the decoding of the kind $\star \to \star$ of `fork` using the kind decoder `decK` in a similar fashion:

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

As another example consider the type `maybe_prod`, defined in Haskell as

```
data MaybeProd a b = NoProd | SomeProd a b
```

This type has kind $\star \to \star \to \star$. We encode it as a type $\Lambda A\,.\,\Lambda B\,.\,1 + A \times B$ 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))).
```

Decoding the type `maybe_prod` gives

```
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 `unit` is the predefined unit type in Coq, and $\times$ and $+$ are the predefined product and sum types.

Finally, we show the code for `apply` : $(\star \to \star) \to \star \to \star$. Its Haskell counter-part is

```
data Apply f a = MkApply (f a)
```

It 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:

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

4 Defining Polytypic Functions

In this section we show how to define polytypic types (Section 4.1) and polytypic functions (Section 4.2) in our framework, and give a number of examples (Sections 4.3 and 4.4). Polytypic functions are defined by giving instances for the type constants in the universe, which we stated in Section 3. We hope 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. We will discuss type and term specialization but we will not formalize these concepts until Sections 5 and 6 respectively.

4.1 Polytypic Types

The type of a polytypic function is a type-level function which, given `np` arguments, constructs a type of kind $\star$. This is represented by the following record:

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

`Record` introduces a record of named fields. `PolyType` depends on one parameter (`np`) and has one field (`typeKindStar`) of type `nary_fn np (decK star) (decK star)`. The term `nary_fn` $n\ A\ B$ denotes the type

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

Argument `np`, known as the *arity* of the polytypic function in Generic Haskell, represents the number of type arguments—it does *not* refer to the number of arguments of the specialized function, which varies with the kind of the target type. Readers familiar with polytypic programming will know that `map` is a polytypic function of arity 2; its type is

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

In Generic Haskell we would give the type `Map` as `A` $\to$ `B`; whereas in Coq we have to explicitly state the type arguments `A` and `B`. Note also that the record constructor `polyType` inherits the arguments to `PolyType`, so the user must provide the number of type arguments `np` as well as the field `typeKindStar`. The type of the polytypic function describes the operation performed at the elements: `map` transforms elements of type `A` to elements of type `B`. Term specialization lifts this operation to structures containing elements and type specialization gives the type of the lifted operation. Informally, type specialization is described as

$$\begin{array}{lcl}
\mathtt{Pt}\langle k\rangle & : & k \to \cdots \to k \to \star \\
\mathtt{Pt}\langle\star\rangle\ T_1 \ldots T_{np} & = & \textit{(user defined)} \\
\mathtt{Pt}\langle k_1 \to k_2\rangle\ T_1 \ldots T_{np} & = & \forall A_1 \ldots A_{np}\ .\ \mathtt{Pt}\langle k_1\rangle\ A_1 \ldots A_{np} \to \mathtt{Pt}\langle k_2\rangle\ (T_1\ A_1) \ldots (T_{np}\ A_{np})
\end{array}$$

This is formalized as a function `specType` (defined in Section 5), so that we can ask Coq to specialize a polytypic type for us. For example, here is the specialization of `Map`$\langle\star \to \star \to \star\rangle$ applied to two copies of the product type:

```
Eval compute in specType tprod Map.
 = ∀ A B : Set, (A → B) →
    ∀ C D : Set, (C → D) → A × C → B × D
```

4.2 Polytypic Functions

To define a polytypic function, the user only needs to provide the definition for the type constants; term specialization takes care of the remaining types. A nice feature of an implementation of polytypic programming in a dependently typed language is that a polytypic function is simply another record which can be passed as an argument to, or computed as the result of, a function.

We define a polytypic function as

```
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
}.
Implicit Arguments PolyFn [np].
```

A polytypic function of `np` arguments has a polytypic type `Pt` of `np` arguments and provides definitions for each of the type constants. The types of these fields are computed by type specialization, ensuring that ill-typed polytypic functions cannot be defined. Moreover, as we have seen, we can ask Coq to evaluate these types, which helps guide us in the construction of a polytypic function. Term specialization is then informally described as

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

where $\lfloor T\rfloor_i$ replaces each free variable A in T by A_i. This definition of `pfn` is formalized as a function `specTerm` (Section 6). For example, specializing `map` (defined in Figure 1) to the datatype `fork` of kind $\star \to \star$ yields:

```
Eval compute in specTerm fork map.
 = fun (A B : Set) (f : A → B) (x : A × A) ⇒
    let (a1, a2) := x in (f a1, f a2)
 : specType fork Map
```

Term specialization returns an actual Coq function, which can be applied to further arguments:

```
Eval compute in specTerm fork map nat nat (fun x ⇒ x + 1) (3, 5).
 = (4, 6)
```

4.3 Examples of Polytypic Types and Functions

As a first example we will consider the polytypic function `count`, which counts the elements in a data structure:

```
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 (plus (f a) (g b)))
(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).
```

Types of kind $\star$ never contain any elements so we simply return 0 when counting units or integers. For other types we pass in a function that converts each element into a natural number, and `count` adds these all up. For example, to count the number of elements in a list we could pass in the constant function that takes each element to the number 1; adding these all up `count` would give us the length of the list. We specialize both `Count` and `count` to the datatype `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'
```

As two further examples, consider `equal` and `less_than`. These functions compare their arguments for equality or check that the first is less than the second. The function `less_than` can be thought of as "pair-wise comparison" or "coordinate-wise order". Lexicographical ordering would perhaps be more sensible on arbitrary data structures, but is a more involved example. We use pair-wise comparison instead to avoid getting bogged down in the details of the example. These two comparison functions are quite similar, and even share the same polytypic type `Compare`:

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

which, specialized to `fork`, yields:

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

Since both `less_than` and `equal` will return false when their arguments have a different structure, the definitions of these two polytypic functions are also quite similar. The only difference is how they compare elements of unit or integer type. The definitions of these two polytypic functions can be found in Figure 4.

In Section 7.4 we will show that even though `less_than` and `equal` are very similar, they do have different properties: we will prove that equality is commutative but `less_than` is anti-commutative.

```
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).
```

```
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).
```

Fig. 4. Polytypic Functions `equal` and `less_than`

4.4 First-class 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. As an example of such a *polytypic combinator* we define "`or_poly`", which computes the "disjunction" of two other polytypic functions of type `Compare`:

```
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 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 y : A + B),
    psum pfn1 A f B g x y || psum pfn2 A f B g x y).
```

Less-than-or-equal-to can now be defined by applying this combinator to the two polytypic functions `equal` and `less_than` we defined above:

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

When we specialize the polytypic function `le` to the type $\mathtt{T} = \Lambda A\,.\,\Lambda B\,.\,A + Z \times B$, we get a function which is provably equal to

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

The proof that these two are equal can be found in `first_class.v` in the Coq sources. Although it is not clear how many higher-order polytypic functions we will be able to define in this fashion, *if* a function can be so defined then the infrastructure for *proofs* we provide in this paper is immediately applicable, which can be very helpful. For instance, although `le` can be defined in Generic Haskell (or indeed using our library) by conjoining the *instances* of `equal` and `less_than` (`le`$\langle T \rangle$ `= equal`$\langle T \rangle$ `|| less_than`$\langle T \rangle$), this does not have the right shape for our proof framework so that the support for polytypic proofs is not available. A further discussion of first class polytypic functions falls outside the scope of this paper.

5 Type Specialization

As we saw in Section 4, the polytypic map function has a polytypic type, and the specialization of map to a type T has a specialized type. In this section we explain how to define type specialization. The full definition of type specialization is given in Figure 5. This section and the next are technical sections which can be skipped by readers who are interested only in the *application* of our infrastructure.

Type specialization is a two-phase process. We first define the kind-indexed type `kit`, where `kit k Map` corresponds to `Pt`$\langle k \rangle$ in the informal definition given in Section 4.1, by induction on `k`. We then apply the result to the appropriate tuple of type arguments.

The case for kind $\star$ is supplied by the user (`PolyType`, see Section 4). We 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. To give a recursive definition of this type, a simple but helpful insight is that it is easier to work with an uncurried form (Altenkirch & McBride, 2003):

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

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.

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

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

```
(* Environment of the form ((a_1, b_1, ..), .., (a_np, b_np, ..))
   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.

(* Specialize polytypic type Pt to kind k *)
Fixpoint kit (k : kind) (np : nat) (Pt : PolyType np)
  : 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.
Implicit Arguments kit [np].

(* Type specialization for open types *)
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).
Implicit Arguments specType' [np nv k ek].

(* 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).
Implicit Arguments specType [np k].
```

Fig. 5. Type Specialization

This definition can be translated to the correct type using the function `quantify_tuple` (defined in `tuples.v` in the Coq sources), which takes a function of the form

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

to the universally quantified type

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

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 (2000) states that specialization of a polytypic function `pfn` of type `Pt` to a type T has type

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

The floor operator $\lfloor T\rfloor_i$ replaces all free variables A in T by A_i. For a closed type T this will have no effect as there are no free variables to replace. To explain this in more detail, let us consider an example: the type `Map` specialized to $T = \Lambda A\,B\,C\,.\,A + B \times C$ should be

$$(A_1 \to A_2) \to (B_1 \to B_2) \to (C_1 \to C_2) \to T\,A_1\,B_1\,C_1 \to T\,A_2\,B_2\,C_2$$

Recall that the polytypic type `Map`, which describes the type of the operations `map` performs at the elements of a structure, is $\Lambda A_1\,A_2\,.\,A_1 \to A_2$. 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 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 6.4), using a function `add_to_ets` to accomplish this. The type of the specialization of the *body* of the lambda abstractions in T 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 this means that all *np* arguments of a polytypic function will be initialized to the *same* type, as most users will only be interested in specializing to closed types.

The full definition of type specialization is given in Figure 5. The function `kit` constructs kind-indexed types and `specType'` returns the application of a kind-indexed type to a tuple $(\lfloor T \rfloor_1, \ldots, \lfloor T \rfloor_{np})$. This tuple of types is constructed by `replace_fvs`, whose definition is straight-forward and can be found in the Coq sources.

6 Term Specialization

Having seen how to define polytypic functions in Section 4 and how to specialize their types in Section 5, we are now in a position to define term specialization. Since the result of term specialization has a specialized type, our implementation is a formal *proof* that the result of term specialization is a term of the type computed by type specialization. The subsections in this section correspond to each of the type constructors for constants, variables, type application and type abstraction.

A polytypic function is fully specified by giving its type and the cases for each of the type constants. The cases for the other types can be inferred; an informal definition of this process $\texttt{pfn}\langle T : k \rangle$ was given in Section 4.2. In this section we discuss its formalization 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
```

The definition is shown in Figure 6; it relies on a number of auxiliary lemmas which we do not show but will explain below (the full definitions can be found in the Coq sources).

```
(* Specialize the polytypic function pfn to open type t *)
Fixpoint specTerm' (np nv : nat) (ek : envk nv) (k : kind)
  (t : type nv ek k) (Pt : PolyType np) (pfn : PolyFn Pt)
  : ∀ (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.
Implicit Arguments specTerm' [np nv ek k Pt].

(* 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.
Implicit Arguments specTerm [np k Pt].
```

Fig. 6. Term Specialization

6.1 Constants

For type constants we have to use the definition provided by the user, but there is a mismatch in the type of the term provided by the user and the return type of term specialization. Consider the case for the product constant. As part of the definition of the polytypic function, the user will have provided a function `pprod` of type

```
pprod : specType tprod Pt
```

Recall from Figure 2 that `tprod` is syntactic sugar for

```
tconst 0 tt tc_prod
```

As described in Section 3, terms of type `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.[2] This arises, for instance, in the use of `tc_prod` in the definition of `fork` in Section 3.4, where instead we have a derivation of the form

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

In general, we need the type encoding

```
tconst nv ek tc_prod
```

for a number of free variables `nv` and associated kind environment `ek`.

We could generalize the definition of the polytypic function given in Section 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 generalization 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 2 (*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 5), we find that we have to prove

$$(\lfloor \texttt{tconst 0 tt tc} \rfloor_1, \ldots) = (\lfloor \texttt{tconst nv ek tc} \rfloor_1, \ldots)$$

This holds as decoding a type constant is independent of the environment provided. □

6.2 Variables

Recall from the informal definition of term specialization (Section 4.2) that in the case for variables we return the function f_A constructed in the clause for lambda abstraction; in the formalization we will use an environment `ef` containing the appropriate function for each

[2] 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.

free variable. The interesting part is to assign a type `envf` to `ef`, since each element in `ef` has a different type. We define `envf` as a heterogeneous tuple (Section 2.6):

```
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 *i*th function is the specialized type of the *i*th 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 simply return the *i*th element in `ef` as the specialized term for variable *i*. However, due to the way we calculate `envf` we do need one technical lemma in the case for variables: `ith_index_f` states that applying a function f to the *i*th element of `elements_of_index` is the same as applying f to i, which boils down to a proof that the *i*th element will always be the index i itself, as `elements_of_index` is a tuple of indices. The *construction* of `ef` will be considered in the case for lambda abstraction (Section 6.4).

6.3 Application

To specialize a polytypic function `pfn` of type `Pt` to a type application (`T U`) we first specialize to $\mathtt{T} : \mathtt{k}_1 \to \mathtt{k}_2$, which will create a term of the form

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

We instantiate the type variables $\mathtt{A}_1 \ldots \mathtt{A}_{np}$ to the elements of the tuple $(\lfloor \mathtt{U} \rfloor_1,\ldots,\lfloor \mathtt{U} \rfloor_{np})$ 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

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

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

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

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

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

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

Lemma 3 (*convert_tapp_specTerm*)

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

Unfolding definitions (Figure 5), we find that we have to prove that

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

This holds as replacing free variables before or after application gives the same result. □

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 (ΛA . T). The type of this specialization must be

$$\texttt{specTerm' (tlam T) pfn ets ef} : \texttt{specType' (tlam T) Pt ets}$$

which can be unfolded to

$$\forall \texttt{A}_1 \ldots \texttt{A}_{np}, \texttt{kit k1 Pt}\,(\lfloor \texttt{tlam T} \rfloor_1, \ldots, \lfloor \texttt{tlam T} \rfloor_{np}) \to$$
$$\texttt{kit k2 Pt}\,(\lfloor \texttt{tlam T} \rfloor_1\ \texttt{A}_1, \ldots, \lfloor \texttt{tlam T} \rfloor_{np}\ \texttt{A}_{np})$$

We will 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.

6.4.1 Dependent currying

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

$$\texttt{fun}\,(\texttt{A}_1, \ldots, \texttt{A}_{np})\,\texttt{f}_{\texttt{A}} \Rightarrow \ldots$$

which we then curry to get

$$\texttt{fun}\,\texttt{A}_1 \ldots \texttt{A}_{np}\,\texttt{f}_{\texttt{A}} \Rightarrow \ldots$$

We cannot use the standard definition of currying, however. The type of the body is

$$\texttt{kit k2 Pt}\,(\lfloor \texttt{tlam T} \rfloor_1\ \texttt{A}_1, \ldots, \lfloor \texttt{tlam T} \rfloor_{np}\ \texttt{A}_{np})$$

and 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
 | 0 ⇒ 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.
Implicit Arguments dep_curry [A n].
```

6.4.2 Specialization to T

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

$$\begin{array}{l}\texttt{specTerm' T pfn}\,(\texttt{add\_to\_ets}\,(\texttt{A}_1,\ldots,\texttt{A}_{np})\,\texttt{ets})(\texttt{add\_to\_ef}\,\texttt{f}_\texttt{A}\,\texttt{ef})\\ \quad :\texttt{specType' T Pt}\,(\texttt{add\_to\_ets}\,(\texttt{A}_1,\ldots,\texttt{A}_{np})\,\texttt{ets})\end{array}$$

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

Lemma 4 (*convert_tlam_specTerm*)

$$\begin{array}{l}\forall\texttt{k2 T Pt ets}\,(\texttt{A}_1,\ldots,\texttt{A}_{np}),\\ \quad\texttt{specType' T Pt}\,(\texttt{add\_to\_ets}\,(\texttt{A}_1,\ldots,\texttt{A}_{np})\,\texttt{ets})\\ =\texttt{kit k2 Pt}\,(\lfloor\texttt{tlam T}\rfloor_1\,\texttt{A}_1,\ldots,\lfloor\texttt{tlam T}\rfloor_{np}\,\texttt{A}_{np})\end{array}$$

Unfolding definitions (Figure 5) we find that we have to prove that

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

This holds as decoding a lambda abstraction and 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. □

6.4.3 Adding f_A to the function environment

The current environment `ef` has an entry for each free variable in `tlam T`, but variable i in `tlam T` becomes variable `Some` i $(i+1)$ in the body `T`. Therefore the function f_X associated with the ith variable X in the old environment:

$$f_X : \texttt{specType' (tvar nv ek i) Pt ets}$$

should have type

$$\begin{array}{l}f_X : \texttt{specType' (tvar (S nv) (k1, ek) (Some i)) Pt}\\ \quad(\texttt{add\_to\_ets}\,(\texttt{A}_1,\ldots,\texttt{A}_{np})\,\texttt{ets})\end{array}$$

in the new environment. The following lemma proves that these two types are equal:

Lemma 5 (*convert_envf*)

$$\begin{array}{l}\forall\,\texttt{nv ek k1 i Pt ets}\,(\texttt{A}_1,\ldots,\texttt{A}_{np}),\\ \quad\texttt{specType' (tvar nv ek i) Pt ets}\\ =\texttt{specType' (tvar (S nv) (k1, ek) (Some i)) Pt}\\ \qquad(\texttt{add\_to\_ets}\,(\texttt{A}_1,\ldots,\texttt{A}_{np})\,\texttt{ets})\end{array}$$

Proof. Unfolding definitions (Figure 5), we find that we have to prove

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

To decode variable i we take the ith element from the environment, which is the same as taking the element `Some` i from an environment containing one extra element. □

When the type of every function in `ef` has been shifted in this way, we can add the argument to the lambda abstraction f_A to the start of `ef`. We need one more lemma:

Lemma 6 (*convert_envf_elem*)

$$\begin{array}{l}\forall\, \texttt{nv ek k1 Pt ets}\ (\texttt{A}_1, \ldots, \texttt{A}_{np}), \\ \quad \texttt{kit k1 Pt}\ (\texttt{A}_1, \ldots, \texttt{A}_{np}) = \texttt{kit k1 Pt}\ (\lfloor \texttt{tvar (S nv) (k1, ek) None} \rfloor_1, \ldots)\end{array}$$

It suffices to prove

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

This is trivially true, because decoding the first variable takes the first element from the type environment, which will always be an element from the tuple of `A`'s. □

7 Polytypic Properties and Proofs

In Sections 3 to 6 we discussed the infrastructure needed for specializing polytypic types and polytypic functions. We now turn to *proofs* over polytypic functions. We discuss how properties of polytypic functions can be stated (Sections 7.1 and 7.2), what polytypic proofs will look like (Section 7.3) and give a few examples (Section 7.4). We conclude with a discussion of alternative formalizations of polytypic properties (Section 7.5); this can be skipped if desired. The formalization of property and proof specialization will be the topic of Sections 8 and 9.

As an introductory example, we will see how to prove that map preserves identity and composition. These two laws are known as the "functor laws", in reference to the laws that a functor must satisfy in category theory. In the case of a unary type constructor, such as $\texttt{fork} : \star \to \star$ (Section 3.4), the functor laws for `map` take the form:

$$\begin{array}{rcl}\text{map}\langle\text{fork}\rangle\ \text{id} & = & \text{id} \\ \text{map}\langle\text{fork}\rangle\ (f \circ g) & = & \text{map}\langle\text{fork}\rangle\ f \circ \text{map}\langle\text{fork}\rangle\ g\end{array}$$

However, given a type constructor of two arguments such as $\texttt{maybe\_prod} : \star \to \star \to \star$, the functor laws take a different shape:

$$\begin{array}{rcl}\text{map}\langle\text{maybe\_prod}\rangle\ \text{id}\ \text{id} & = & \text{id} \\ \text{map}\langle\text{maybe\_prod}\rangle\ (f \circ g)\ (h \circ k) & = & \text{map}\langle\text{maybe\_prod}\rangle\ f\ h \circ \text{map}\langle\text{maybe\_prod}\rangle\ g\ k\end{array}$$

The shape of these properties therefore depends on the kind of the datatype we specialize to. Fortunately, we can state and prove such properties in much the same way as we type and define polytypic functions.

We have seen how we can formally interpret the informal notation $pfn\langle T\rangle$ 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. 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 previously.

7.1 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 property is that the function must itself be the identity function:

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

Like stating polytypic types, stating polytypic properties is not always trivial. The intuition for this example is that given identity functions $f_1, \ldots, f_n$, $(\mathtt{map}\ f_1\ \ldots\ f_n)$ must also be an identity function. In the degenerate case where there are no f_i, i.e. the case for kind $\star$, we simply have that `map` must be the identity.

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

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

From the definition of the type of the property and the case for kind $\star$, we are able to derive the property for other kinds. The informal definition of property specialization is very similar to that of type specialization given in Section 5:

$$\begin{array}{lcl} \mathtt{Pp}\langle\star\rangle\ T_1 \ldots T_{nt} & = & \textit{(user defined)} \\ \mathtt{Pp}\langle k_1 \to k_2\rangle\ T_1 \ldots T_{nt} & = & \lambda (f_1, \ldots, f_{nx})\ .\ \forall A_1 \ldots A_{nt} : k_1\ .\ \forall (g_1, \ldots, g_{nx})\ . \\ & & \quad \mathtt{Pp}\langle k_1\rangle\ (A_1, \ldots, A_{nt})\ (g_1, \ldots, g_{nx}) \to \\ & & \quad \mathtt{Pp}\langle k_2\rangle\ (T_1\ A_1, \ldots, T_{nt}\ A_{nt})\ (\mathtt{app\_fs}\ (f_1, \ldots, f_{nx})\ (g_1, \ldots, g_{nx})) \end{array}$$

For example, the instance of `Id` for kind $\star \to \star$ will be:

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

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

$$\forall (A : \star)\ (g : A \to A)\ .\ (\forall y : A\ .\ g\ y = y) \to \forall x : T\ A\ .\ \mathtt{map}\langle T\rangle\ A\ A\ g\ x = x$$

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

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

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

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

As before, the definition of the property for other kinds can now be derived. For example:

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

The property applied to three instances of $\mathtt{map}\langle T\rangle$ will then be

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

This is a generalization of the usual property, which we obtain by instantiating g_3 by $g_1 \circ g_2$. We *need* to generalize the property because in $(\mathtt{map}\ f) \circ (\mathtt{map}\ g) = \mathtt{map}(f \circ g)$ we have three different instantiations of `map`: once with f, once with g and once with $f \circ g$.

7.2 Polytypic Properties, Formally

We define a polytypic property using the following record type:

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

The record contains two fields: `idxs` provides information about the type of the property, and `propKindStar` gives the property for kind $\star$. The record is dependent on four arguments, where `np` is implicit in the type of `Pt`:

		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

Hopefully two examples will go a long way towards clarifying this definition. The property that map preserves identity is stated using our library in Coq as

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

Similarly, the property that map preserves composition is 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).
```

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 might be added to the Coq library as well, but we have left this to future work for now.

Let us have a look at the type of `Pp`: given *nt* type arguments $T_1 \dots T_{nt}$, the type of a polytypic property indexed by a kind k generally looks like

$$\texttt{Pp}\langle k\rangle\ T_1 \dots T_{nt} : \texttt{Pt}\langle k\rangle\ \underbrace{(T_1, \dots, T_{nt})}_{1} \times \cdots \times \texttt{Pt}\langle k\rangle\ \underbrace{(T_1, \dots, T_{nt})}_{nx} \to \texttt{Prop}$$

where $\underbrace{(T_1, \dots, T_{nt})}_{i}$ takes the correct *np* type arguments for the ith occurrence of `Pt` from the tuple of type arguments $(T_1, \dots, T_{nt})$ 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 of type arguments is given by `idxs` in the description of the polytypic property. This representation of polytypic properties limits their expressiveness somewhat, as they can only refer to a single polytypic type. This makes it impossible to state the interaction between a polytypic encoder and decoder, for example. A generalization should not be too difficult, but is left as future work.

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

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

The type of polytypic properties is less straight-forward than the type of polytypic types, which was simply $k \to \cdots \to k \to \star$. Therefore our choice of record fields might also be a little less obvious. In Section 7.5 we explain why we use `idxs` to determine the type of `Pp` rather than asking the user for the type directly.

7.3 Polytypic Proofs

When we define a polytypic 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 should be regarded as a formal proof that providing proofs for the type constants is indeed sufficient. The definition of a polytypic proof mirrors the definition of a polytypic function (Section 4.2):

```
Lemma map_Comp : PolyProof map Comp.
Proof.
  (* split into subgoals and unfold definitions *)
  apply (polyProof map Comp); compute;
  (* tint and tunit solved automatically *)
  auto; intros.
  (* tprod: apply hypothesis about the components of the pair (H, H0) *)
  destruct x ; rewrite H ; rewrite H0 ; auto.
  (* tsum: apply appropriate hypothesis (H for inl, H0 for inr)  *)
  destruct x ; [rewrite H | rewrite H0] ; auto.
Defined.
```

Fig. 7. Example polytypic proof: Map preserves composition

```
Record PolyProof (nt nx np : nat) (Pt : PolyType np)
  (pfn : PolyFn Pt) (Pp : PolyProp nt nx Pt) : Type :=
 polyProof {
  prfUnit : specPropTo tunit Pp pfn ;
  prfInt  : specPropTo tint  Pp pfn ;
  prfProd : specPropTo tprod Pp pfn ;
  prfSum  : specPropTo tsum  Pp pfn
}
Implicit Arguments PolyProof [nt nx np Pt].
```

In a polytypic proof we state the polytypic function `pfn` and the property `Pp` we want to prove, and give proofs for each of the type constants. Figure 7 gives an example: the proof that map preserves composition. To be able to understand the proof the reader needs to know about Coq's tactic language Ltac, and run the proof in Coq. The details of the proof are beyond the scope of this paper. The important point is that the proof is short and easy.

The polytypic proof that map preserves identity is very similar, and it should be possible to write a Coq tactic (proof search algorithm) to automate parts of these proofs. We have left this to future work.

Informally, proof specialization is defined as:

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

This will be formalized in Section 9. Here we show that we are now able to prove that map specialized to `fork` preserves composition simply by applying proof specialization to the lemma `map_Comp`:

```
Lemma mapCompFork : specPropTo fork Comp map.
Proof (specProof fork map_Comp).
```

7.4 Polytypic Property and Proof Examples

In this section we will show that while the two comparison functions defined in Section 4.3 share the same type, they do not necessarily have to satisfy the same properties as well. We will prove that `equal` is commutative while `less_than` is anti-commutative.

The polytypic property that describes commutativity of polytypic functions of type `Compare` is defined as:

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

We want to prove that this property holds for `f` instantiated to `equal⟨T⟩`:

$$\forall\, \mathtt{x}\ \mathtt{y} : T, \mathtt{equal}\langle \mathtt{T}\rangle\ \mathtt{x}\ \mathtt{y} = \mathtt{equal}\langle \mathtt{T}\rangle\ \mathtt{y}\ \mathtt{x}$$

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

```
Eval compute in specPropTo fork Commutative equal.
= ∀ (A : Set) (f : A → A → bool), (∀ 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
```

To prove commutativity of equality we need to establish the property for the type constants. This is the importance of the following lemma, whose proof can be found in the file `examples_polyprop.v` in the Coq sources.

```
Lemma eq_Comm : PolyProof equal Commutative.
```

We have shown that the polytypic function `equal` is commutative, but the opposite property holds for `less_than`: if $x < y$ then not $y < x$. Anti-commutativity is represented by the polytypic property `AntiCommutative`:

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

Specializing `AntiCommutative` to `fork` and applying it to instances of `less_than` gives us the property we would expect:

```
Eval compute in specPropTo fork AntiCommutative less_than.
= ∀ (A : Set) (f : A → A → bool),
    (∀ x y : A, f x y = true → f y x = false) →
    ∀ x y : A * A, less_than⟨fork⟩ A f x y = true →
    less_than⟨fork⟩ A f y x = false
```

To prove that `less_than` is anti-commutative we need to prove the lemma `lt_AntiComm`:

```
Lemma lt_AntiComm : PolyProof less_than AntiCommutative.
```

The proof of this lemma, as well as some additional examples of polytypic properties and their proofs can be found in the Coq sources.

7.5 Alternative Definitions

To specify a property using our formalization, the user must give the type of the property by means of the `idxs` tuple of tuples of indices and the property for kind $\star$. This mechanism for specifying the type of the property may not be the most obvious choice, so in this section we give the rationale for choosing this particular approach.

In the definition of a polytypic type (`PolyType`, Section 4.1) we do not ask the user to specify the kind of the polytypic type. We do not need to because it has a very simple form:

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

Unfortunately, properties are more complicated: as mentioned in Section 7.2, 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 is 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 8). Intuitively, the more leeway we give to the user, the less we are able to 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 function `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 the shuffling of type arguments 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 definition of a polytypic property does not give us enough information for property specialization. In particular, when specializing the property to 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. However, when we leave the shuffling of type parameters to the case for kind $\star$ this information is not available and we cannot instantiate f.

8 Property Specialization

Section 7.2 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 $k_1 \to k_2$ is then derived using property specialization. In this section, we discuss the formal definition of property specialization.

The informal definition of property specialization was given in Section 7.1. When we compare the informal definition of property specialization to that of type specialization (Section 4.1), 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 $\star \to \star$ (`Comp`, Section 7.1):

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

To prove preservation of composition for the polytypic function `map` specialized to `fork` we will instantiate the property as follows: $nx = 3$, the functions in the tuple (f_1, f_2, f_3) will all be instantiated to $\texttt{map}\langle\texttt{fork}\rangle$, and the tuple (g_1, g_2, g_3) corresponds to the three functions in the informal statement of the property. The statement

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

corresponds to the application of $\texttt{map}\langle\texttt{fork}\rangle$ to each of (g_1, g_2, g_3). The function `app_fs` is however not quite simple application. The types of each f_i and g_i are

$$f_i : \texttt{Pt}\langle k_1 \to k_2\rangle\ \underbrace{(T_1, \ldots, T_{nt})}_i$$
$$g_i : \texttt{Pt}\langle k_1\rangle\ \underbrace{(A_1, \ldots, A_{nt})}_i$$

From Section 5 we know that a polytypic type specialized to an arrow kind $k_1 \to k_2$ takes the form

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

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

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

We see that the argument expected here matches the type of g_i exactly, so we apply the term 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).

```
(* Specialize polytypic property Pp to kind k *)
Fixpoint kip (k : kind) (nt nx np : nat) (Pt : PolyType np)
 (Pp : PolyProp nt nx Pt) : ∀ 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.
Implicit Arguments kip [nt nx np Pt].

(* Property specialization for open types *)
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 k Pp (replace_fvs t ets).
Implicit Arguments specProp' [nt nx np nv k ek Pt].

(* Property specialization for closed types *)
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).
Implicit Arguments specProp [nt nx np k Pt].

(* Property specialized applied to the correct tuple *)
Definition specPropTo (nt nx np : nat) (k : kind)
  (t : closed_type k) (Pt : PolyType np) (Pp : PolyProp nt nx Pt)
  (pfn : PolyFn Pt) : Prop :=
 specProp t Pp (cst_closed t pfn (idxs Pp)).
Implicit Arguments specPropTo [np nt nx k Pt].
```

Fig. 8. Property Specialization

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

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

This construction of property specialization and its application to a number of type arguments follows type specialization (Section 5) exactly. The corresponding Coq definition is given as `specProp'` in Figure 8 and like `specType`, `specProp` instantiates `specProp'` to closed types.

We now have a way to specialize properties to a particular type. For example, we can specialize the property `Id` of preservation of identities to the integer type:

```
Eval compute in specProp tint Id.
= fun fns : (Z → Z) × unit ⇒
    ∀ x : Z, let f := fst fns in f x = x
```

When we start doing proofs, we want to prove this property for a particular instantiation of the tuple `fns`. Given a polytypic property, a polytypic function and the type we want to specialize it to, the functions `cst_closed` and `cst` create the appropriate tuple to be applied.

We have added an abstraction called `specPropTo` (see Figure 8 for the definition) which takes a type, property, and polytypic function and returns the specialized property applied to the appropriate tuple of functions. For example, the property that `map` preserves identities specialized to the type for integers is evaluated by the expression:

```
Eval compute in specPropTo tint Id map.
= ∀ (x : Z), x = x
```

9 Proof Specialization

Having seen how to do property specialization in Section 8, we now discuss how to do proof specialization. Since proof specialization constructs a proof of the property defined by property specialization, our formalization of proof specialization is a formal proof that giving a proof for each of the type constants is indeed sufficient. As for term specialization, the subsections in this section correspond to the type constructors for constants, variables, application and abstraction.

The informal definition of proof specialization was given in Section 7.3. This definition of proof specialization is very similar to the informal definition of term specialization that we gave in Section 4.2, except that proofs need an additional tuple of arguments

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

corresponding to the specialized instances of the polytypic function for which we want to prove the property.

In the informal definition of proof specialization many details are omitted. In particular, since T might be an open type, 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 environment 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 5, the type of each function argument $\mathtt{pfn}\langle T\rangle_j$ ($1 \leq j \leq nx$) requires a similar mapping for each of its *np* type arguments; this environment is given by $\underbrace{(\mathtt{ets})}_j$.

`efs` As explained in Section 6, each function argument $\mathtt{pfn}\langle T\rangle_j$ requires an environment `ef` containing functions for the free variables in T; `efs` is a tuple of *nx* such environments, one for each argument $\mathtt{pfn}\langle T\rangle_j$.

```
(* 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).
Implicit Arguments envp [nt nx np nv ek Pt ets].

(* 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.
Implicit Arguments specProof' [nt nx np nv k ek Pt pfn Pp].

(* 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.
Implicit Arguments specProof [nt nx np k Pt pfn Pp].
```

Fig. 9. 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 9 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. 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 the type `T`; `specProof` simply calls `specProof'` with the appropriately constructed empty environments.

9.1 Constants

The case for constants is given by the user except that, as in Section 6.1, we need a weakening lemma. This weakening lemma is a good example of dealing with heterogeneous equalities, so we will spell the proof out in some detail.

Lemma 2 in Section 6.1 proved that

$$\texttt{Pt}\langle k\rangle\ (\lfloor \emptyset \vdash C : k\rfloor_1, \ldots, \lfloor \emptyset \vdash C : k\rfloor_n) = \texttt{Pt}\langle k\rangle\ (\lfloor \Gamma \vdash C : k\rfloor_1, \ldots, \lfloor \Gamma \vdash C : k\rfloor_n)$$

We proved 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, i.e. the decoding of C. The specialization of a polytypic function for a type constant then is the definition given by the user converted using the proof p of the above equality:

$$\texttt{convert}\ p\ \textit{(user definition)}$$

For proof specialization we have to prove a similar lemma:

Lemma 7 (`convert_tconst_specProof`)
$\texttt{Pp}\langle k\rangle\ (\lfloor \emptyset \vdash C : k\rfloor_0, \ldots)\ (\texttt{pfn}\langle \emptyset \vdash C : k\rangle, \ldots) = \texttt{Pp}\langle k\rangle\ (\lfloor \Gamma \vdash C : k\rfloor_0, \ldots)\ (\texttt{pfn}\langle \Gamma \vdash C : k\rangle, \ldots)$

We again show that the two argument tuples are the same. We already proved this about the first argument tuples as part of Lemma 2; 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 8

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

The specialization of a polytypic function to a type constant simply returns the definition that was given by the programmer converted using a weakening lemma (Lemma 2). Hence, both sides of the equality reduce to

$$\begin{array}{ll} & \texttt{convert}\ (\text{Lemma 2 at } \emptyset)\ \textit{(user definition)} \\ \simeq & \texttt{Pt}\langle k\rangle\ (\lfloor \emptyset \vdash C:k\rfloor_0,\ldots),\texttt{Pt}\langle k\rangle\ (\lfloor \Gamma \vdash C:k\rfloor_0,\ldots) \\ & \texttt{convert}\ (\text{Lemma 2 at } \Gamma)\ \textit{(user definition)} \end{array}$$

which follows from Lemma 1. We then prove Lemma 7 using the method that we sketched in Section 2.7: generalize over Lemma 8, rewrite with Lemma 2, and complete the proof.

9.2 Variables

Recall from Section 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 6), the difficulty in specializing the case for variables is the definition of the type of the environment, in this case `ep`. Informally, the ith element in `ep`, corresponding to the proof for the ith variable, has type

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

The formal definition of the type of `ep`, called `envp`, is given in Figure 9. The *construction* of `ep` will be considered when we discuss lambda abstraction in Section 9.4.

9.3 Application

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

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

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 the tuple $(A_1,\ldots,A_{nt})$ by $(\lfloor U\rfloor_1,\ldots,\lfloor U\rfloor_{nt})$ and the tuple $(g_1,\ldots,g_{nx})$ by the tuple $(\mathtt{pfn}\langle U\rangle_1,\ldots,\mathtt{pfn}\langle U\rangle_{nx})$ in $\mathtt{IH}_T$ and apply the result to $\mathtt{IH}_U$ we get a term of type

$$\begin{aligned}&\mathtt{Pp}\langle k_2\rangle\ (\lfloor T\rfloor_1\ \lfloor U\rfloor_1,\ldots,\lfloor T\rfloor_{nt}\ \lfloor U\rfloor_{nt})\\ &\ (\mathtt{app\_fs}\ (\mathtt{pfn}\langle T\rangle_1,\ldots,\mathtt{pfn}\langle T\rangle_{nx})\ (\mathtt{pfn}\langle U\rangle_1,\ldots,\mathtt{pfn}\langle U\rangle_{nx}))\end{aligned}$$

To get the type we actually need, we specify two conversion lemmas. The first conversion is fairly straight-forward, and is a corollary of Lemma 3 in Section 6.3.

Corollary 1

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

The second lemma is more interesting:

Lemma 9 (*convert_tapp_specProof*)

$$\begin{aligned}\forall\ T\ U,\quad &(\mathtt{app\_fs}\ (\mathtt{pfn}\langle T\rangle_1,\ldots,\mathtt{pfn}\langle T\rangle_{nx})\ (\lfloor U\rfloor_1,\ldots,\lfloor U\rfloor_{nt}))\\ &\simeq_{(\mathtt{Pt}\langle k_2\rangle\ (\underbrace{\lfloor T\rfloor_1\lfloor U\rfloor_{1,\ldots}}_{1})\ \times\ \cdots),(\mathtt{Pt}\langle k_1\rangle\ (\underbrace{\lfloor T\,U\rfloor_{1,\ldots}}_{1})\ \times\ \cdots)}\\ &(\mathtt{pfn}\langle T\ U\rangle_1,\ldots,\mathtt{pfn}\langle T\ U\rangle_{nx})\end{aligned}$$

The proof involves some manipulation of heterogeneous equalities. Note that Corollary 1, in addition to proving the first argument tuples equal, also proves that the two types involved in the heterogeneous equality in Lemma 9 are equal. □

9.4 Lambda Abstraction

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

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

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

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

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

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

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

- A set of *nt* types, given by `ets`, used to define $\lfloor T \rfloor_i$ $(1 \leq i \leq nt)$.
- 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`.

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. Hence we need to convert the types of the proofs in the `ep` environment to

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

where the decoding is now interpreted with respect to the extended environment `ets`.

This involves proving that for each variable i of kind k, we have:

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

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

Since there are quite a few calls to conversion lemmas hidden here, proving this lemma involves a lot of reasoning about various heterogeneous equalities. In fact, we had to change the formalization of term specialization to apply converts at a smaller granularity before we were able to prove this lemma (see also Section 2.7).

Once all environments have been extended we need to apply the induction hypothesis $\texttt{IH}_T$, but first we will need two conversion lemmas to get a proof of the correct type. The first lemma is a corollary of Lemma 4 in Section 6.4:

Corollary 2

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

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 10 (*convert_tlam_specProof*)

$$\begin{aligned}&\texttt{app\_fs}\ (\texttt{pfn}\langle \Lambda A\ .\ T\rangle_1, \ldots, \texttt{pfn}\langle \Lambda A\ .\ T\rangle_{nx})\ (\texttt{pfn}\langle A\rangle_1, \ldots, \texttt{pfn}\langle A\rangle_{nx})\\ &\simeq_{(\texttt{Pt}\langle k_1 \to k_2\rangle\ (\underbrace{\lfloor \Lambda A\ .\ T\rfloor_1 A_{1,\ldots}}_{1})\ \times\ \cdots),(\texttt{Pt}\langle k_1 \to k_2\rangle\ (\underbrace{\lfloor T\rfloor_{1,\ldots}}_{1})\ \times\ \cdots)}\\ &(\texttt{pfn}\langle T\rangle_1, \ldots, \texttt{pfn}\langle T\rangle_{nx})\end{aligned}$$

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

10 Recursion

We have come a long way. We showed how to define polytypic types and functions in Section 4, how to do type and term specialization in Sections 5 and 6, how to specify properties of polytypic functions and give polytypic proofs in Section 7 and how to specialize properties and proofs in Sections 8 and 9.

However, so far we have not discussed recursive datatypes. Unfortunately, we cannot simply add recursion directly to our type universe, as Hinze does (2000): 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. Moreover, since all functions in Coq must be total, the specialization of polytypic functions must also be total.

In this section we discuss how to define corecursive polytypic functions in our framework. Since datatypes in Haskell have both finite and infinite inhabitants we will focus on corecursion rather than recursion. Unfortunately, corecursion 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.

Like our universe, the universe in Generic Haskell or Generic Clean does not include a general recursion operator either, as discussed by Löh (2004) in Section 7.5.1 and Alimarine (2005) in 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 decode `list_struct` 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 $(\texttt{fromList}, \texttt{toList})$ where we prove that $\texttt{fromList} \circ \texttt{toList} = \texttt{id} = \texttt{toList} \circ \texttt{fromList}$.

Given such a structural representation for lists, we 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[3]

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

We then 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)).
```

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 must terminate, and Coq guarantees termination 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. Calls must be "guarded-by-constructors" (Bertot, 2005). Guardedness can be described in two steps (Bertot & 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.

[3] The function `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.

A corecursive function is guarded if all of its 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.

However, the only reason that it does not pass Coq's guardedness check is because Coq does not sufficiently unroll its definition. 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
```

Intuitively, this can be reduced to

```
match l with
| nil ⇒ nil
| cons a l' ⇒ cons (f a) (mapList f l')
end
```

and we see that `mapList` *is* in fact guarded. Unfortunately, the Coq guardedness checker is not sophisticated enough to see it.

10.1 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.[4]

We must modify `mapList'` slightly so that it does not return the resulting `list' B` directly, but instead applies a continuation `K`. We then move the application of `K` into the branches of the match construct, ensuring that Coq will unfold the application of `K`.

```
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 :=
```

[4] Thanks to Bruno Barras and Russell O'Connor on the Coq mailing list for this suggestion.

```
 match l with
 | inl u ⇒ K (inl _ u)
 | inr (a, l') ⇒ K (inr _ (f a, rec f l'))
 end.
Implicit Arguments CPS_mapList' [A B R].
```

We then 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).
```

The `@`-notation in `@toList B` allows us to implicitly state the type arguments to `toList` which would normally be explicit. 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 the use of CPS transforms enables Coq to sufficiently unroll the term to verify that it is guarded, the integration of this method with our existing development causes some problems. In particular, we would need to modify the return type of term specialization to incorporate the CPS transform.

As discussed in Section 6 the type of *term* specialization is given by *type* specialization:

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

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 lists and the definition of polytypic types and functions given by the user must change. Moreover, 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
```

whereas we would expect a continuation passing version of map to have the type

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

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.

10.2 μ-Reduction

Coq checks guardedness with regards to the input modulo β (function application), δ (unfolding of constants), ι (pattern matching and unrolling of fixpoints) and ζ (local definitions) reduction. 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, similar to the *case-of-case transformation* implemented in the Glasgow Haskell Compiler (see Section 5 of (Peyton Jones, 1996)). 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 $(p\,\texttt{nil})$, and the branch for the `cons` constructor will be a binary function of the form $\lambda a\ l\,.\,e'$, where $e' : p\,(\texttt{cons}\ a\ l)$.

Figure 10 shows the definition of μ-reduction. It makes use of an operator $f \bullet_n g$ which computes 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 the $(\bullet)$-operator to a branch, n will be the arity of the

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

$$\begin{array}{ll} & \mathrm{match}_p\Big(\mathrm{match}_{p'}\ e' \text{ with } \langle f'_1,\ldots\rangle\Big) \text{with } \langle f_1,\ldots\rangle \\ \to_\mu & \mathrm{match}_{p\bullet p'}\ e' \text{with } \langle(\lambda x\,.\,\mathrm{match}_p\ x \text{ with } \langle f_1,\ldots\rangle)\bullet f'_1,\ldots\rangle \end{array}$$

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

Fig. 10. μ-Reduction

constructor associated with that branch; in our examples we will leave n implicit. The unrolled definition of `mapList` that we saw above is represented internally in Coq as

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

Applying μ-reduction to this term, as well as the other reductions, results in the term

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

which corresponds to the definition of `mapList` that we would expect.

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. The *diff* file detailing our modifications to the Coq source code can be found online (Verbruggen, 2009).

10.3 Proofs

In Section 10.2, we have shown how coinduction can be used to define the specialization of a polytypic function to a corecursive 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: `map_Id`. Since we are dealing with potentially infinite lists we can prove the functor laws only up to bisimilarity, which we define as

```
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`. We now define `mapIdList'`, a proof that the function `mapList'` preserves identities, given that we have such a proof for the tail of the list:

```
Lemma 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 `map_Id` to `list'`. However, since unlike for equality we cannot give one general definition of bisimilarity that can be used on all datatypes, we cannot even *state* the polytypic property, much less give a polytypic proof for it. This is also the reason that we use equality rather than bisimilarity for the *elements* of the list. 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 corecursion only.

Given a proof of preservation of identity for `list'`, the proof of preservation of identity for `list` 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))
```

Although the actual proof is a little more complicated because we need to manually unroll coinductive definitions, the proof as shown is the proof "up to some matching on equality". We can see that the proof follows the structure of `mapList` exactly; the only difference is that the result is not guarded by `toList` but by `to_preserves_bisim_list`, which is defined as

```
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 10.2.

10.4 Reflection

Adding μ–reduction to the guardedness checker allows us to define `map` and various other examples. This shows that at least in principle it is possible to extend the guardedness checker in such a way that we can use the Generic Haskell approach to recursion. However, checking productivity with a syntactic guardedness check that relies on unrolling is computationally expensive and remains brittle. For example, adding μ–reduction is not sufficient to check the guardedness of coinductive proofs computed by proof specialization, although it is not clear at this point what is stopping the guardedness checker from unrolling the proof sufficiently. Ultimately, a semantic approach, possibly type based (Abel, 2006; Abel, 2009), would be preferable but such approaches are not currently available for Coq.

11 Related Work

The literature on generic programming is vast and a detailed survey of approaches to generic programming is beyond the scope of this paper. We refer the reader to the survey paper by Hinze *et al.* (2006), or to 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 limit the discussion to related work on *formal* (that is, machine verified) definitions of and proofs about generic functions.

11.1 PolyP-style Generics

The work by Pfeifer & Rueß (1999) is 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 is known in the functional programming community as origami programming (Gibbons, 2006). The translation to a dependently typed language is reasonably straight-forward and we do not reproduce it here. However, note that definitions such as `map` and `fold` as they are usually given in PolyP-style of programming

```
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
```

cannot be used in a dependently typed language that uses structural recursion. 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 `map` function can then of course easily be defined.

The second part of the paper formalizes the syntactic approach familiar from PolyP (Jansson & Jeuring, 1997). The universe that is considered is simpler than the universe in PolyP, however, and contains only sums, products, and two distinguished free variables to denote the type argument and the recursive occurrences of the datatype.

Like in PolyP, a syntactic approach is used to define `bimap` once and for all for every datatype that can be represented in the universe. Moreover, the bimap laws—preservation of identity and composition—are proven, and the proofs are trivial.

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 the work by Pfeifer & Rueß (1999); 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 first universe, they define fold exactly as described above as "PolyP style"; in a type theory such as Coq that relies on "obvious" structural induction, such a definition will not be accepted.

The authors continue by extending the first 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.

Compared to the universe we consider in this paper, however, the class of kinds supported by both is limited to first-order kinds (in the case of Pfeifer & Rueß (1999), even only kind $\star \to \star$, like in PolyP).

11.2 Strictly Positive Types

Morris *et al.* (2006) find a way to decode a universe of strictly positive types that includes recursion. The approach relies on a cunning idea: rather than giving a direct translation from a code to a datatype, the authors give a datatype `El` 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. We will discuss why we cannot adopt this approach in Section 12.

When we compare their universe to the universe we use in this paper (Figures 2 and 3) we find many differences. Some seem merely cosmetic; for example, the type constants and the constructors for selecting specific free variables have been inlined into the definition of the universe. The authors say that inlining the constructors for selecting free variables makes proofs easier and avoids some conversion lemmas; it would be interesting to see if it would have the same effect in our universe. Other differences are more important: in particular, the universe does not include type application, and therefore covers only polymorphic types of first order kind.

Polytypic functions are functions on elements of the decoder datatype `El`. Polytypic functions for this universe are quite similar to their Generic Haskell counterpart, except in how they deal with free variables. It should however be possible to abstract this out in a similar way that we have done in this paper, by defining a record comparable to `PolyFn` and a separate specialization process.

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 this paper. 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.

The authors continue (2009) with an extension to dependent datatypes, but a discussion of these "strictly positive families" is beyond the scope of this paper.

11.3 Generic Haskell-style Generics

There are a few implementations of Generic-Haskell style generics in a dependent programming language: Altenkirch & McBride (2003) give an implementation in the language Oleg (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 do any proofs, polytypic or otherwise, over polytypic programs.

Since both our implementation in Coq 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 the 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.

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 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, polytypic proofs can also be smaller. Finally, it will make the formalization more accessible to programmers used to Generic Haskell.

11.4 Containers

Containers (Hoogendijk & de Moor, 2000; Abbott *et al.*, 2003) and also (Jay, 1995; Backhouse & 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.*, 2005) is one notable exception and uses type theory instead.

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: a characterization of the *shape* of the term, a mapping from this shape to a set of *positions* within the term, and a mapping from these positions to the values in the term.

The standard example is the container encoding of a list: the shape of the list is a natural number, the positions in a list of shape n is represented by the set of indices $\{0 \ldots n-1\}$, and the mapping from positions to values is the list indexing operation.

Some definitions and proofs about containers can be done semantically. For instance, we can easily prove (up to eta-expansion) that every container is a functor. 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 & Altenkirch, 2009). A discussion of indexed containers is, however, beyond the scope of this paper.

12 Future Work

As discussed in Section 9 a polytypic proof is defined by providing the proofs for each of the type constants. Since these proofs are relatively straight-forward and all follow a similar structure, it should be possible to automate such polytypic proofs somewhat.

Quite a bit of machinery is necessary before we can specialize polytypic functions and proofs to coinductive datatypes. This machinery includes a structural representation and a corresponding embedding-projection pair for each datatype, the actual definition of the polytypic function specialized to the coinductive type, and a number of standard lemmas for each datatype. Most of this infrastructure is also needed in Generic Haskell, where it is generated by the Generic Haskell preprocessor. It would be useful to provide tactics that can automatically generate these definitions.

There are a number of ways in which our system can be extended. We would like to extend the type of polytypic properties to allow properties concerning multiple polytypic functions of different type, such as the property that the polytypic `decode` function is right inverse to the polytypic `encode` function. Further, to allow for polytypic functions such as parsers and pretty printers our universe needs to be extended to include some meta-information such as constructor names.

Another interesting extension would be the use of type-indexed types. Since we are working in a dependently typed language one would hope that there is no essential difference between type-indexed values and type-indexed types. Using type-indexed types we can state the preservation of identity law 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.

Unfortunately, the way we set up the universe in Section 3.2 requires that the codomain of a polytypic function must be a type in an impredicative universe. This means that supporting type-indexed types may be more difficult than it should be: by choosing `Set` we cannot return a type from a polytypic function and we cannot simply substitute `Type` for `Set` as it is not impredicative.

In Section 10.4 we mention that we cannot use proof specialization if we want to construct proofs over coinductive datatypes. The proofs returned by proof specialization are too complex for the Coq guardedness checker to deal with and it will not be able to verify that proofs constructed from these definitions are guarded. Some careful dissecting of the result of proof specialization is required to find the cause for this guardedness problem.

The ideal solution to dealing with recursion would be to extend our universe directly with support for recursion. This extension is a challenging research problem, however. In particular, it is difficult to see how we could adopt the approach taken in the definition of the universe of strictly positive types (see Section 11.2). As explained, Morris *et al.* (2006) 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.

Finally, there are of course semantic differences between Generic Haskell and Coq which means that proofs over Generic Haskell-style programs using our framework do not *quite* describe the same programs as the programs in Generic Haskell itself. We feel that the framework we provide will nevertheless be useful to Generic Haskell programmers, but alternative solutions are also possible. For instance, we might embed Haskell as a domain specific language within Coq and reason about its semantics explicitly. This would however be a considerable amount of work and quite a different approach to the one we advocate in this paper.

13 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 formalize type specialization within the host language we get type checking of polytypic functions virtually for free. Further, polytypic functions can be reified within the host language 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 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.

Acknowledgements

We would like to thank the anonymous reviewers for their detailed comments which improved the paper considerably.

References

Abbott, Michael, Altenkirch, Thorsten, & Ghani, Neil. (2003). Categories of containers. *Pages 23–38 of: Proceedings of the 6th international conference on foundations of software science and computation structures, Warsaw, Poland.* Lecture Notes in Computer Science, vol. 2620. Springer–Verlag.

Abel, Andreas. (2006). *Type-based termination: A polymorphic lambda-calculus with sized higher-order types*. Ph.D. thesis, Fakültat für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München.

Abel, Andreas. (2009). Type-based termination of generic programs. *Science of computer programming*, **74**(8), 550–567. Special Issue on Mathematics of Program Construction (MPC'06).

Alimarine, Artem. (2005). *Generic functional programming: Conceptual design, implementation and applications*. Ph.D. thesis, Radboud Universiteit Nijmegen, The Netherlands.

Altenkirch, Thorsten, & McBride, Conor. (2003). Generic programming within dependently typed programming. *Pages 1–20 of: Proceedings of the ifip tc2/wg2.1 working conference on generic programming, Schloss Dagstuhl, July 2002*. Kluwer, B.V.

Altenkirch, Thorsten, McBride, Conor, & McKinna, James. 2005 (April). *Why dependent types matter*. Manuscript, available online. `http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf`, Sep 12, 2010.

Backhouse, Roland, & Hoogendijk, Paul. (2003). Generic properties of datatypes. *Pages 97–132 of: Generic programming: Advanced lectures*. Lecture Notes in Computer Science, vol. 2793. Springer–Verlag.

Benke, Marcin, Dybjer, Peter, & Jansson, Patrik. (2003). Universes for generic programs and proofs in dependent type theory. *Nordic journal of computing*, **10**(4), 265–289.

Bertot, Yves. (2005). Filters on coinductive streams, an application to eratosthenes' sieve. *Pages 102–115 of: TLCA'05: Proceedings of the 7th international conference on typed lambda calculi and applications, Nara, Japan*. Lecture Notes in Computer Science, vol. 3461. Springer–Verlag.

Bertot, Yves, & Castéran, Pierre. (2004). *Coq'Art: Interactive theorem proving and program development*. Springer–Verlag.

Bertot, Yves, & Komendantskaya, Ekaterina. (2008). Inductive and coinductive components of corecursive functions in Coq. *Electronic notes in theoretical computer science*, **203**(5), 25–47.

Coq Development Team. (2008a). *Coq frequently asked questions (v8.1)*. `http://www.lix.polytechnique.fr/coq/node/16`, Sep 12, 2010.

Coq Development Team. (2008b). *Coq reference manual (version 8.2)*. `http://www.lix.polytechnique.fr/coq/refman/`, Sep 12, 2010.

de Bruijn, N. G. (1972). A lambda calculus notation with nameless dummies. *Indagationes mathematicae*, **34**, 381–392.

Gibbons, Jeremy. (2006). Datatype-generic programming. *Pages 1–71 of: School on datatype-generic programming*. Lecture Notes in Computer Science, vol. 4719. Springer–Verlag.

Harper, Robert, & Pollack, Robert. (1991). Type checking with universes. *Theoretical computer science*, **89**, 107–136.

Hinze, Ralf. (2000). *Generic Programs and Proofs*. Habilitationsschrift, Universität Bonn, Germany.

Hinze, Ralf. (2006). Generics for the masses. *Journal of functional programming*, **16**, 451–482.

Hinze, Ralf, & Löh, Andres. (2006). Generic programming, now! *Pages 150–208 of: School on datatype-generic programming*. Lecture Notes in Computer Science, vol. 4719. Springer–Verlag.

Hinze, Ralf, & Löh, Andres. (2009). Generic programming in 3D. *Science of computer programming*, **74**, 590–628.

Hinze, Ralf, & Peyton Jones, Simon. (2001). Derivable type classes. *Electronic notes in theoretical computer science*, **41**(1), 227–236.

Hinze, Ralf, Jeuring, Johan, & Löh, Andres. (2006). Comparing approaches to generic programming in Haskell. *Pages 72–149 of: School on datatype-generic programming*. Lecture Notes in Computer Science, vol. 4719. Springer–Verlag.

Hoogendijk, Paul, & de Moor, Oege. (2000). Container types categorically. *Journal of functional programming*, **10**(2), 191–225.

Hurkens, Antonius J. C. (1995). A simplification of Girard's paradox. *Pages 266–278 of: TLCA'95: Proceedings of the second international conference on typed lambda calculi and applications, Edinburgh, UK*. Springer–Verlag.

Jansson, P., & Jeuring, J. (1997). PolyP—a polytypic programming language extension. *Pages 470–482 of: Popl'97: Conference record 24th ACM SIGPLAN–SIGACT symposium on principles of programming languages, Paris, France*. ACM Press.

Jay, C. Barry. (1995). A semantics for shape. *Pages 251–283 of: Selected papers of ESOP'94, the 5th european symposium on programming*. Elsevier Science Publishers B. V.

Lämmel, Ralf, & Peyton Jones, Simon. (2003). Scrap your boilerplate: A practical design pattern for generic programming. *Pages 26–37 of: TLDI'03: ACM SIGPLAN international workshop on types in language design and implementation, New Orleans, Louisiana, USA*, vol. 38. ACM Press.

Lämmel, Ralf, & Visser, Joost. (2002). Typed combinators for generic traversal. *Pages 137–154 of: PADL'02: Proceedings of the 4th international symposium on practical aspects of declarative languages, Portland, OR, USA*. Lecture Notes in Computer Science, vol. 2257. Springer–Verlag.

Löh, Andres. (2004). *Exploring generic Haskell*. Ph.D. thesis, Instituut voor Programmatuurkunde en Algoritmiek, Utrecht, The Netherlands.

McBride, Conor. (1999). *Dependently typed functional programs and their proofs*. Ph.D. thesis, University of Edinburgh.

McBride, Conor. (2002). Elimination with a motive. *Pages 197–216 of: TYPES'00: Selected papers from the international workshop on types for proofs and programs, Durham, UK*. Springer–Verlag.

Morris, Peter, & Altenkirch, Thorsten. (2009). Indexed containers. *LICS'09: 24th IEEE symposium in logic in computer science, Los Angeles, California, USA*.

Morris, Peter, Altenkirch, Thorsten, & McBride, Conor. (2006). Exploring the regular tree types. *Pages 252–267 of: TYPES'04: Types for proofs and programs, France*. Lecture Notes in Computer Science, vol. 3839. Springer–Verlag.

Morris, Peter, Altenkirch, Thorsten, & Ghani, Neil. 2007 (January). Constructing strictly positive families. *CATS'07: The australian theory symposium, Ballarat, Australia*.

Morris, Peter, Altenkirch, Thorsten, & Ghani, Neil. (2009). A universe of strictly positive families. *International journal of foundations of computer science*, **20**(1), 83–107.

Norell, Ulf. (2002). *Functional generic programming and type theory*. M.Phil. thesis, Computing Science, Chalmers University of Technology.

Peyton Jones, Simon. (1996). Compiling Haskell by program transformation: A report from the trenches. *Pages 18–44 of: ESOP'96: Proceedings of the european symposium on programming, Linköping, Sweden*. Lecture Notes in Computer Science, vol. 1058. Springer–Verlag.

Pfeifer, Holger, & Rueß, Harald. (1999). Polytypic proof construction. *Pages 55–72 of: TPHOLs'99: Proceedings of the 12th international conference on theorem proving in higher order logics, Nice, France*. Springer–Verlag.

Plotkin, Gordon. (1975). Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, **1**, 125–159.

Rodriguez, Alexey, Jeuring, Johan, Jansson, Patrik, Gerdes, Alex, Kiselyov, Oleg, & d. S. Oliveira, Bruno C. (2008). Comparing libraries for generic programming in Haskell. *Pages 111–122 of: Haskell'08: Proceedings of the first ACM SIGPLAN symposium on Haskell, Victoria, British Columbia, Canada*. ACM Press.

Rodriguez, Alexey, Holdermans, Stefan, Löh, Andres, & Jeuring, Johan. (2009). Generic programming with fixed points for mutually recursive datatypes. *Pages 233–244 of: ICFP'09: Proceeding of the 14th ACM SIGPLAN international conference on functional programming, Edinburgh, UK*. ACM Press.

Schrijvers, Tom, Peyton Jones, Simon, Sulzmann, Martin, & Vytiniotis, Dimitrios. (2009). Complete and decidable type inference for GADTs. *Pages 341–352 of: ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, Edinburgh, UK*. ACM.

Sheard, Tim. (2007). Generic programming in Ωmega. *Pages 258–284 of: Spring school on datatype-generic programming*. Lecture Notes in Computer Science, vol. 4719. Springer–Verlag.

Sørensen, M. H., & Urzyczyn, P. (2006). *Lectures on the Curry-Howard isomorphism*. Elsevier.

Verbruggen, Wendy. (2009). *Coq sources*. `http://www.wendyverbruggen.net/publications`.

Vytiniotis, Dimitrios, Weirich, Stephanie, & Peyton Jones, Simon. (2006). Boxy types: Inference for higher-rank types and impredicativity. *Pages 251–262 of: ICFP'06: Proceedings of the 11th ACM SIGPLAN international conference on functional programming, Portland, Oregon*. ACM Press.