Polytypic Programming in Coq

Wendy Verbruggen* Edsko de Vries Arthur Hughes
Computer Science Department, Trinity College Dublin, Ireland
{verbrw, devriese, aphughes}@cs.tcd.ie

Abstract
The aim of our work is to provide an infrastructure for formal proofs over Generic Haskell-style polytypic programs. For this goal to succeed, we must have a definition of polytypic programming which is both fully formal and as close as possible to the definition in Generic Haskell. In this paper we show a formalization in the proof assistant Coq of type and term specialization. Our definition of term specialization can be interpreted as a formal proof that the result of term specialization has the type computed by type specialization.

Categories and Subject Descriptors D.2.4 [Software/Program Verification]: Formal methods

General Terms Languages, Theory, Verification

Keywords Polytypic programming, generic programming, theorem proving, kind-indexed types, Coq, formalization

1. Introduction
The aim of our work is the development of an infrastructure in the proof assistant Coq (Bertot and Castéran 2004) for doing proofs over polytypic programs in the style of Generic Haskell (Löh 2004) or Generic Clean (Alimarine 2005) which can be used by practitioners of these systems. This paper gives a formalization of polytypic programming in Coq and is therefore an important first step towards our goal.

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

Although there are many approaches to generic programming both in dependently typed languages and in more conventional functional programming languages (Section 7), few support the kind-indexed types which characterize Hinze’s work. The core idea is that if f is a polytypic function of type Φ we can specialize f to an ordinary function f(T) over a datatype T. The type of f(T) is the specialization F(T) to the kind of T. Term specialization (f(T)) is defined by induction on the structure of T; type specialization (F(T)) is defined by induction on the kind of T.

To illustrate this approach, we will consider the polytypic map function with type Map (we will give the definitions of map and Map in Section 2). The specialization of map to the type of integers is simply the identity on integers:

Eval compute in specTerm tint map.

= fun (z:Z) ⇒ z : specType tint Map

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

Eval compute in specType tint Map.

= Z → Z : Set

as expected (Set is Coq’s name for kind ⋆).

As a second more interesting example consider the type tfork, defined as Λα. (α, α). To map a function across a term of this type we need a function to map across its elements. Thus the specialization of map to tfork is

fun (A B:Set) (f:A → B) (x:A * A) ⇒
let (a, b) := x in (f a, f b) : specType tfork Map

Similarly, to map across a term of type tprod, Λα. Λβ. (α, β), we need two functions to map its elements. The specialization of map to tprod is

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) : specType tprod Map

Finally, the specialization of map to tapply, Λφ. Λα. φ α, requires two functions to translate the datatype and its elements.

fun (T T’:Set → Set)
(f:∀ (A B:Set), (A → B) → T A → T’ B)
(A B:Set) (g:A → B) ⇒ f A B g : specType tapply Map

The point of these various examples is to show that we can specialize map to types of any kind. For kind (tint) we get the identity function. For first-order kinds such as * → * or * → * → * (tfork and tprod) we get a function which maps functions at the base types (f and g) across the datatype. Finally, even higher-order kinds such as (∗ → ∗) → ∗ → *(tapply) are supported.
The aim of this work is to provide an implementation of polytypic programming in Coq which is easily recognizable to programmers familiar with Generic Haskell or Generic Clean. In particular, our contributions are:

- We provide an infrastructure for defining polytypic functions and their types which is very similar to the infrastructure provided by Generic Haskell or Generic Clean (Section 2).
- We formalize term specialization and type specialization in Coq as defined in Generic Haskell/Clean. In particular, the definition of our type universe is identical (modulo syntactic differences).
- The definition in Coq has one very important benefit over the existing implementation in Generic Haskell. Since we use dependent types to specify that the result of a polymorphic function must be of the form \( \text{specTerm} \ T \ f \) must be of the form \( \text{specType} \ T \ F \), our implementation is a formal proof that the term specialization \( f(T) \) must have type \( F(T) \).

The final point is important since it paves the way towards our ultimate goal of providing an infrastructure in Coq to prove properties about Generic Haskell style programs. For this goal to succeed, we must have a definition of polytypic programming which is both fully formal and as close as possible to the definition in Generic Haskell or Generic Clean. This paper provides such a definition; Section 2 shows that the interface we provide to programmers is very similar to the Generic Haskell interface. After a brief introduction to Coq in Section 3, we give the definition of the generic view in Section 4 followed by the formalization of type and term specialization in Sections 5 and 6.

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

## 2. Defining polytypic functions

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

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

\[
\text{Record PolyType} (\text{np:}\text{nat}) : \text{Type} := \text{polyType} \{ \text{typeKindStar} \}.
\]

The Record keyword introduces a record of named fields; the difference between records in Coq and records in Haskell is that records in Coq can be dependent. We will take a closer look at dependent types in the next section; suffice to say here that a dependent type is one that depends on a term (rather than a type). PolyType has one parameter \( np \) and one field \( \text{typeKindStar} \) of type \text{nary_fn} np Set Set. A term \text{nary_fn} \ n \ A \ B denotes the type

\[
\underbrace{A \rightarrow \ldots \rightarrow A} \rightarrow B
\]

We will refer to \( np \) as the number of arguments of the polytypic function (it does not refer to the number of arguments of the specialized function, which varies with the kind of the target type—see previous section).

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

\[
\text{Definition Map} : \text{PolyType} \ 2 := \text{polyType} \ 2 \ (\text{fun} \ A \ B \Rightarrow A \rightarrow B).
\]

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

To define a polytypic function, the user only needs to provide the definition for the type constants; term specialization then takes care of the remaining types. A nice feature of an implementation of polytypic programming in a dependently typed language is that the definition of a polytypic function is simply another record. Polytypic functions are therefore first-class (ordinary objects in the host language) and can be passed as arguments or returned as results. We define a polytypic function as

\[
\text{Record PolyFn} (\text{np:}\text{nat}) : \text{Type} := \text{polyFn} \ { \text{ptype} : \text{PolyType} \ np ; \text{pint} : \text{specType} \ \text{tunit} \ \text{ptype} ; \text{pprod} : \text{specType} \ \text{tprod} \ \text{ptype} ; \text{psum} : \text{specType} \ \text{tsum} \ \text{ptype} }.
\]

In words, a polytypic function of \( np \) arguments has a (polytypic) type of \( np \) arguments, and contains definitions for the type constants \text{tunit}, \text{tint}, \text{tprod} and \text{tsum} (for simplicity’s sake, we do not consider other type constants). The type of these fields is determined by type specialization (explained in Section 5), which ensures that users cannot define ill-typed polytypic functions. For the specific case of \text{Map}, this simplifies to

\[
\text{Definition map} : \text{PolyFn} \ 2 := \text{polyFn} \ { \text{fun} \ (u:\text{unit}) \Rightarrow u ; \text{fun} \ (z:\text{Z}) \Rightarrow z ; \text{fun} \ (A \ B:\text{Set}) \ (f:A \rightarrow B) \Rightarrow \text{let} \ (a, c) := x \in (f \ a, g \ c) \text{ in } \text{match} x \text{ with} | \text{inl} \ a \Rightarrow \text{inl} \ D \ (f \ a) | \text{inr} \ c \Rightarrow \text{inr} \ B \ (g \ c) \text{ end} }.
\]

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

\[
\lambda f \rightarrow \lambda g \rightarrow \lambda x \rightarrow \text{let} \ (a, c) = x \in (f \ a, g \ c)
\]

To conclude this section, we will consider another classic polytypic function: polytypic equality. Unlike \text{map}, \text{equal} only has a single argument. Its type is defined as

\[
\text{Definition equal} : \text{PolyType} \ 1 := \text{polyType} \ 1 \ (\text{fun} \ A \ B \Rightarrow A \rightarrow B).
\]
The calculus of constructions (Coquand and Huet 1988) is a depen-
dence of constructions (higher-order predicate logic) extended with

3.1 Dependent types

Before we delve into our formalization of polytypic programming, we will first give a brief overview of Coq. Coq is a proof assistant
developed in Inria (Bertot and Castéran 2004) based on the calcu-
lus of constructions (higher-order predicate logic) extended with

Although it may be slightly unusual to define \texttt{tupleT} recursively while
defining \texttt{Fin} inductively, we found this the most convenient setup. An
alternative is to define tuples functionally. This removes the need for an
indexing operator and makes some of the lemmas we need redundant;
howerver, it also introduces some new lemmas and operations and, more
importantly, it makes it impossible to prove that two tuples are equal.

\footnote{Although it may be slightly unusual to define \texttt{tupleT} recursively while
defining \texttt{Fin} inductively, we found this the most convenient setup. An
alternative is to define tuples functionally. This removes the need for an
indexing operator and makes some of the lemmas we need redundant;
howerver, it also introduces some new lemmas and operations and, more
importantly, it makes it impossible to prove that two tuples are equal.}
To understand this error, we must know a little more about universes in Coq. Like in Haskell, the natural number “5” has type nat. Like in Haskell, the type nat has kind (or type) Set (set is called * in Haskell). Unlike in Haskell, however, this hierarchy continues ad infinitum: Set has type Type₀, Type₀ has type Type₁, and generally Typeᵢ has type Typeᵢ₊₁. Moreover, there is a coercion rule that if T : Typeᵢ, then T : Typeⱼ for any j ≥ i. This stratification of type prevents the encoding of logical paradoxes (e.g., Hurkens 1995).

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

tupleT : Typeᵢ → nat → Typeⱼ (i ≤ 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 Typeⱼ. The constructed type itself must then have type

tupleT : Typeᵢ → Typeⱼ

Since we pass Typeⱼ as the first argument to tupleT of type Typeᵢ, we must have Typeⱼ : Typeᵢ which will hold if j < i. But the constraints i ≤ j and j < i cannot both be satisfied, and Coq reports a universe inconsistency: there is no suitable assignment that does not result in an inconsistency.

The problem is that Coq does not support universe polymorphism (Harper and Pollack 1991). A work-around would be to duplicate the definition of tupleT. 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 []). Given a tuple

(x, y, z)

of elements of some type A, we construct the type

\( f x \times f y \times f z \)

This is implemented as

```
Fixpoint gtupleT (A:Set) (n:nat) (f:A → Typeᵢ) :
  tupleS A n : Typeᵢ :=
  match n return tupleS A n → Type with
  | 0 ⇒ fun tup := unit
  | S m ⇒ fun tup ⇒
    let (t, ts) := tup in f t * gtupleT m f ts end.
```

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

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

3.3 Proofs

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

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

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

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

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

The two definitions of MP are indistinguishable. Coq does however make a formal distinction between terms that are “programs” and terms that are “proofs” in the type system. The type of a program (say, nat) lives in Set, as we saw above. The type of a proof (that is, a proposition such as 1 = 1) lives in Prop (both Set and Prop live in Type₀). The reason for the two different sorts is that Coq supports program extraction: Coq can extract all the computational content (that is, keep the programs but strip the proofs) to be exported to OCaml or Haskell for efficient compilation.

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

We will not make use of tactics in this paper, so we refer the reader to (Bertot and Cast´eran 2004) for more information. However, the support for tactics and proof automation is an important reason for choosing Coq for our work (reasoning about polytypic programs).

4. Definition of the Generic View

A generic view is a set of codes that represent the datatypes that can be used as a target for specialization of polytypic functions. For example, if we want to specialize the polytypic map function to the product type (map prod in Coq), we must pass the code for prod (tprod), as well as the definition of map itself, as arguments to term specialization. However, the result of term specialization should be a function on the product datatype proper (prod). This means that we must define a mapping from codes in the generic view to ordinary Coq datatypes. Such a mapping is known as a decoder. The definitions for our generic view and the decoders are listed in Figure 1.

4.1 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-kindned types can be represented. An element

\( T : \text{type } nv ek k \)

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

\( ek ⊢ T : k \)

The type of the environment \( ek \) is \( \text{envk } nv \), which is an \( nv\)-tuple of kinds.
(* Codes for kinds *)
Inductive kind : Set :=
| star : kind
| karr : kind → kind → kind.

(* 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:nat) (ek:envk nv) (k:kind), type_constant k → type nv ek k
| tvar : ∀ (nv:nat) (ek:envk nv) (i:Fin nv), type nv ek (getS i ek)
| tapp : ∀ (nv:nat) (ek:envk nv) (k1 k2:kind), type nv ek (karr k1 k2) → type nv ek k1 → type nv ek k2
| tlam : ∀ (nv:nat) (ek:envk nv) (k1 k2:kind), type (S nv) (k1, ek) k2 → type nv ek (karr k1 k2).

(* Syntactic sugar for types with no free variables *)
Definition closed_type (k:kind) : Set :=
| tconst 0 tt tc_unit.
| tconst 0 tt tc_int.
| tconst 0 tt tc_prod.
| tconst 0 tt tc_sum.

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

(* Decoder for 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) {struct t} : envt nv ek return decK k :=
match t in type nv ek k return decK k with
| tconst nv ek k tc ⇒ fun et ⇒ match tc in type_constant k return decK k with
| tc_unit ⇒ unit
| tc_int ⇒ Z
| tc_prod ⇒ prod_set
| tc_sum ⇒ sum_set
end.

(* Example: Λ(α : *) · α × α.*)
Definition tfork :=
(tlam (tapp (tapp (tconst 1 (star,tt) tc_prod)
| tvar 1 (star, tt) (fz 0))
| tvar 1 (star, tt) (fz 0))).

(* Example: Λ(α : * → *) · Λ(β : *) · α β.*)
Definition tapply :=
(tlam (tapp 2 (star, (karr star star, tt)) (fs (fz 0))
| tvar 2 (star, (karr star star, tt)) (fz 1))).

Figure 1. Generic View and Decoders
For example, the rule for lambda abstractions encodes the kind-

\[(k_1, ek) : T : k_2\]  
\[ek : \Delta T : k_1 \rightarrow k_2 \quad \text{LAM}\]

\footnote{Set was impredicative in Coq by default before version 8; this was changed mainly to support classical reasoning. We will not use classical reasoning, however, and so making Set impredicative does not compromise the soundness of our proofs (see Coq Development Team 2008).}

Note that we are using De Bruijn indices to represent variables (de Bruijn 1972). The indices in a type of \(n_v\) free variables are of type \(\text{Fin}~n_v\), which guarantees that no indices can be out of bounds.

### 4.2 Decoding kinds

The decoder for kinds is straightforward, but there is a subtlety with the choice of \(\text{Set}\) as the decoding of kind \(\star\). In the specialization of the arrow kind (Section 5), we will construct types of the form

\[(\forall (\alpha : \text{decK} \text{ star}) \ldots) : \text{decK} \text{ star}\]

Since the bound variable \(\alpha\) ranges over the very type that is defined, the type of \(\alpha\) must be impredicative. As explained in Section 3, Type in Coq is not impredicative (but stratified) and returning Type for the decoding of kind \(\star\) will result in a universe inconsistency when we subsequently attempt to define type specialization. Hence we must choose \(\text{Set}\) instead, enabling the impredicative \(\text{Set}\) option\(^2\). It does not seem possible to give a formalization of Generic Haskell-style polytypic programming without using an impredicative universe.

### 4.3 Decoding types

The decoder for types is more involved. To decode a type \(T\) with \(n_v\) free variables, we must know the decoded types of the free variables in \(T\). Hence, we need an environment \(\text{envT}\) of type \(\text{envT}\) that associates a decoded type \(T_i\) with every free variable \(i\) in \(T\). Since the type of \(T_i\) (its kind, if you prefer) depends on the kind of \(T\), each element in \(\text{envT}\) has a different type. We therefore calculate \(\text{envT}\) from the kind environment \(ek\): \(\text{envT} = \text{gtupleT} \text{ decK} \text{ ek}\).

Using the generalized product described in Section 3.

We have already introduced two different environments \((ek : \text{envK} n_v \text{ and } et : \text{envT} n_v ek)\) and we will need two more in the remainder of the paper. As it may be difficult to keep track of so many different environments we provide an overview of the definitions and their purpose in Figure 2.

Armed with this environment we can now define the decoder for types as shown in Figure 1. Type constants map to their Coq definitions and their purpose in Figure 2. Type definitions and their purpose in Figure 3.

Figure 1 also shows the encoding for two of the example types that we considered in the introduction: \(t\text{fork}\) and \(t\text{apply}\). The result of decoding these types is

\[
\text{decT} \quad \text{tfork tt} = \text{fun (arg:Set)} \Rightarrow \text{arg} \cdot \text{arg} :: \text{decK (karr star star)}
\]

\[
\text{decT} \quad \text{tapply tt} = \text{fun (f:Set} \rightarrow \text{Set}) (a:Set) \Rightarrow f \cdot a :: \text{decK (karr (karr star star) (karr star star))}
\]

The full definition is shown in Figure 3.

Type specialization is a two-phase process. We first define the kind-indexed type \(kit k Map\). Hinze (2000a) denotes this by \(\text{Poly}(k)\) for some polytypic type \(\text{Poly}\) and defines it by induction on \(k\):

\[
\text{Poly}(k :: [] :: \ldots :: k) \Rightarrow k \rightarrow \ldots \rightarrow k \rightarrow \star
\]

\[
\text{Poly}(\ast) T_1 \ldots T_{n_p} = \text{(user defined)}
\]

\[
\begin{align*}
\text{Poly}(k_1 : k_2) T_1 \ldots T_{n_p} = & \forall A_1 \ldots A_{n_p}. \text{Poly}(k_1) A_1 \ldots A_{n_p} \rightarrow \text{Poly}(k_2) (T_1 A_1) \ldots (T_{n_p} A_{n_p}) \\
\text{Poly}(k_1) (A_1, \ldots, A_{n_p}) & \Rightarrow \text{Poly}(k_2) (T_1 A_1, \ldots, T_{n_p} A_{n_p})
\end{align*}
\]

The case for kind \(\star\) is the user-defined type (\(\text{PolyType}, \text{Section 2.5}\)).

We can rewrite the case for arrow kinds as

\[
\begin{align*}
\Lambda(T_1, \ldots, T_{n_p}) & \cdot \forall A_1 \ldots A_{n_p}. \text{Poly}(k_1) (A_1, \ldots, A_{n_p}) \Rightarrow \text{Poly}(k_2) (T_1 A_1, \ldots, T_{n_p} A_{n_p}) \\
\text{Poly}(k_1) (A_1, \ldots, A_{n_p}) & \Rightarrow \text{Poly}(k_2) (T_1 A_1, \ldots, T_{n_p} A_{n_p})
\end{align*}
\]

This function can be implemented as follows:

\[
\begin{align*}
\text{Fixpoint quantify_tuple (A:Type) (n:nat) : \text{tupleT} A n \rightarrow \text{Set} :=}
\end{align*}
\]

\[
| \text{match n return (tupleT A n} \rightarrow \text{Set} \rightarrow \text{Set}} \rightarrow \text{Set with}
\]

\[
| 0 \Rightarrow \text{fun f} \Rightarrow f \cdot tt
\]

\[
| \text{S m} \Rightarrow \text{fun f} \Rightarrow \forall a : A, \text{quantify_tuple A m (fun As} \Rightarrow f (a, As))
\]

\[
\text{end}
\]

Paraphrasing, \(kit\) constructs a type that calculates the required specialized type given a tuple \((T_1, \ldots, T_{n_p})\); the second step in type specialization is therefore to construct this tuple. Hinze states that specialization of a polytypic function \(\text{poly}\) of type \(\text{Poly}\) to a type \(T\) has type

\[
\text{poly}(T :: k) :: \text{Poly}(k) ([T]_1, \ldots, [T]_{n_p})
\]

\footnote{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.}
The definition of the floor operator \( \lfloor \cdot \rfloor \) is slightly involved, so we will consider an example first\(^4\). The type of map specialized to the datatype \( T = \Lambda A B C . \ A + B \times C \) 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 of map, which describes the type of the operations maps performs at the elements of a structure, is \( \Lambda A B . \ A \to B \). When we specialize map to a specific datatype, we will need an instance of this operation for each of the arguments of that datatype. Hence if the datatype has \( n_v \) parameters, we will need \( n_v \) copies of this operation, each of which will need \( n_v \) type arguments. To keep track of all of these types, we construct an environment \( ea : \text{enva} \) of the form

\[
\langle (A_1, \ldots, A_{n_v}), (B_1, \ldots, B_{n_v}), (C_1, \ldots, C_{n_v}), \ldots \rangle
\]

The floor function \( \lfloor \cdot \rfloor \), replaces each free variable in \( T' \) (each argument of the datatype) by the \( 't \) variable associated with it by creating the tuple \( (A_1, B_1, C_1, \ldots) \) and then decoding \( T' \) using this new tuple as type environment (Section 4.3).

Returning to our example, for every \( \Lambda A . \ldots \) we encounter during term specialization we will add the correct tuple \( (A_1, \ldots, A_{n_v}) \) to \( ea \) (Section 6.4). The type of the specialization of the \( \text{body} \) of the lambda abstractions in \( T \) will then be

\[
\text{Poly}(k) ([A + B \times C]_1, \ldots, [A + B \times C]_{n_v})
\]

\(^4\)Hinze uses naming conventions to define the floor operator, but unfortunately naming conventions do not work in a formal setting.

When we specialize a function to a closed type (\( n_v = 0 \)), \( ea \) must be empty and \( ([T]_1, \ldots, [T]_{n_p}) \) degenerates to \( (T, \ldots, T) \). From a user’s perspective (who will always specialize polytypic functions to closed types), this means that all \( n_p \) arguments of a polytypic function will be initialized to the \textit{same} type (see also Section 2).

The full definition of type specialization is shown in Figure 3; it calculates kind-indexed types and \textit{specType} returns the application of a kind-indexed type to a tuple \( ([T]_1, \ldots, [T]_{n_p}) \). This tuple is created by \textit{replace_fvs}, whose definition is straightforward and can be found in the Coq sources (Verbruggen 2008).

6. Term specialization

A polytypic function is fully specified by giving its polytypic type and the cases for all constants. The cases for all other types can be inferred. Hinze (2000a) gives the definition for the specialization of a polytypic function \textit{poly} of type \textit{Poly} to a type \( T :: k \) as

\[
\text{poly}(T :: k) :: \text{Poly}(k) [T]_1 \ldots [T]_{n_p}
\]

\[
\text{poly}(C :: kC) = \langle \text{user defined} \rangle
\]

\[
\text{poly}(A :: kA) = f_A
\]

\[
\text{poly}(\Lambda A . \ T :: k_1 \to k_2) = \lambda A_1 \ldots A_{n_p} . \text{poly}(T :: k_2)
\]

\[
\text{poly}(T U :: k_2) =
\]

\[
\text{poly}(T :: k_1 \to k_2) \ [U]_1 \ldots [U]_{n_p} \ (\text{poly}(U :: k_1))
\]

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

\[
\text{specTerm} : \forall (n p : \text{nat}) (k : \text{kind}) (t : \text{closed_type} k) (pf : \text{PolyFn} n p), \text{specType} t (\text{ptype} pf)
\]
Since \texttt{specTerm} returns a term of the type computed by \texttt{specType}, the definition of \texttt{specTerm} is a formal proof that term specialization returns terms of the required type. The definition of term specialization is shown in Figure 4; it relies on a number of auxiliary lemmas which we do not show but will explain below. As always, the full definitions can be found in the Coq sources. In the remainder of this section we will describe each of the clauses in the definition of \texttt{specTerm}.

### 6.1 Constants

The case for type constants seems straightforward. After all, we should simply use the definition given by the user. But there is a subtlety we must deal with. Consider the case for the product constructor, \texttt{tprod}.

As described in Section 4, instances of type encode kinding derivations; \texttt{tprod} encodes the derivation in the empty environment

\[
\emptyset \vdash \texttt{tconst } tt \texttt{tc} \texttt{prod} : \star \rightarrow \star 
\]

When \texttt{tc} \texttt{prod} is used inside another type, however, it may well be used in an environment where there are free variables. This arises, for instance, in the use of \texttt{tc} \texttt{prod} in the definition of \texttt{tfork} in Figure 1, where instead we have a derivation of the form

\[
\alpha : \star \vdash \texttt{tconst} tt \texttt{tc} \texttt{prod} : \star \rightarrow \star 
\]

Generally, we need a function of type

\[
\texttt{specType'} \ (\texttt{tconst} \nv \ek \texttt{tc} \texttt{prod}) \ptype \ea
\]

for some number of free variables \texttt{nv} and associated kind environment \texttt{ek} (\texttt{ea} is the environment we need for the type arguments in the generated type, and will be discussed later). We could generalize the definition of the polytypic function to

\[
\text{Record PolyFn} \ (\np : \text{nat}) : \text{Type} := \text{polyFn} \{
\ptype : \text{PolyType} \np ;
\pprod : \forall \ (\np : \text{nat}) \ (\ek : \text{envk} \nv \ek) \ (\text{ea} : \text{enva} \ \np \ek),
\text{specType'} \ (\texttt{tconst} \nv \ek \texttt{tc} \texttt{prod}) \ptype \ea ;
\ldots
\}\n\]

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

**Lemma 1 (weakening \texttt{tconst}).** For all \texttt{nv}, \texttt{tc}, \texttt{ek}, \texttt{Pt}, \texttt{ea},

\[
\text{specType} \ (\texttt{tconst} 0 \texttt{tt} \texttt{tc}) \ \texttt{Pt}
\]

is the same type as

\[
\text{specType'} \ (\texttt{tconst} \texttt{nv} \texttt{ek} \texttt{tc}) \ \texttt{Pt} \ \texttt{ea}
\]

**Proof.** Unfolding definitions (Figure 3), we find that we have to prove

\[
(\texttt{tconst} 0 \texttt{tt} \texttt{tc})_i, \ldots = (\texttt{tconst} \texttt{nv} \texttt{ek} \texttt{tc})_j, \ldots
\]

The equalities between the elements are trivial, so we can complete the proof by induction on the length of the tuples (\np).

### 6.2 Variables

Recall from the definition of term specialization as given by Hinze (2000a) that in the case for variables we return the function \texttt{fA} constructed in the clause for lambda abstraction. However, Hinze’s definition relies on naming conventions which do not translate to a formal setting. Instead we need an environment \texttt{ef} with an entry for each of these functions.

The tricky part is to assign a type \texttt{envf} to \texttt{ef}, since each element in \texttt{ef} has a different type. We can compute \texttt{envf} using the generalized tuple from Section 3 as follows:

\[
gtupleS \ (\texttt{fun} \ i \Rightarrow \text{specType'} \ (\texttt{tvar} \nv \ek \ i) \ \texttt{Pt} \ \texttt{ea}) \ (\text{elements_of_fin} \ \nv).
\]

The type of the \texttt{i}’th function is the specialized type of the \texttt{i}’th free variable. Thus, we map \texttt{specType} over the tuple \((0, 1, \ldots, \texttt{nv} - 1)\) constructed by \texttt{elements_of_fin}. Given \texttt{ef} we can simply return the \texttt{i}’th element in \texttt{ef} as the specialized term for variable \texttt{i}.

The construction of \texttt{ef} will be considered in the case for lambda abstraction (Section 6.4).

### 6.3 Application

To specialize a polytypic function \texttt{pf} of type \texttt{Pt} to a type application \((\texttt{T U})\) we first specialize to \(\texttt{T} : k_1 \rightarrow k_2\), which will create a term of the form

\[
\text{specTerm'} \ T \ \texttt{pf} \ \texttt{ea} \ \texttt{ef} : \forall \ A_1 \ldots A_\np, \\kit k1 \ Pt \ (A_1, \ldots, A_\np) \rightarrow \kit k2 \ Pt \ (\ [T]_1 A_1, \ldots, [T]_\np A_\np).
\]

We instantiate the type variables \(A_1, \ldots, A_\np\) in \texttt{specTerm} \(T\) \texttt{pf} \texttt{ea} \texttt{ef} to the elements of the tuple \((\ [U]_1, \ldots, [U]_\np)\) using

\[
\text{intstantiate_tuple} \ (X : \text{Type}) :\ (n : \text{nat}) :\ \forall \ (\text{args : ttupleT X n}) : (X : \text{tupleT X n} \rightarrow \text{Set}), \\quad \text{quantify_tuple} X X \text{args}
\]

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

\[
\texttt{specTerm'} \ T \ \texttt{pf} \ \texttt{ea} \ \texttt{ef} : \forall \ A_1 \ldots A_\np : \kit k1 \ Pt \ (\ [U]_1, \ldots, [U]_\np) \rightarrow \kit k2 \ Pt \ (\ [T]_1 [U]_1, \ldots, [T]_\np [U]_\np).
\]

We can apply this to the specialized term of \texttt{U}, which serendipitously has exactly the right type, and get a term of type

\[
\kit k2 \ Pt \ (\ [T]_1 [U]_1, \ldots, [T]_\np [U]_\np)
\]

Since we are specializing an application, the return type we expect here would be

\[
\texttt{specType'} \ (\texttt{tapp} \ T \ U) \ \texttt{Pt} \ \texttt{ea}
\]

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

**Lemma 2 (convert_tapp).** For all \np k1 k2 \texttt{ea} \texttt{Pt} and types \(T : k1 \rightarrow k2 \texttt{and U} : k1, \texttt{the type}

\[
\kit k2 \ Pt \ (\ [T]_1 [U]_1, \ldots, [T]_\np [U]_\np)
\]

is the same type as

\[
\texttt{specType'} \ (\texttt{tapp} \ T \ U) \ \texttt{Pt} \ \texttt{ea}
\]

\footnote{\texttt{gtupleS} is a version of \texttt{gtupleT} that returns a \texttt{Set} rather than a \texttt{Type}.}

\footnote{Due to the way we calculate \texttt{envf}, we do need one technical lemma \(\texttt{(ith_fin)}\) that the \(i\)’th element of \texttt{elements_of_fin} is \texttt{i}.}
(* Term specialization *)

Fixpoint specTerm' (np nv:nat) (ek:envk nv) (k:kind) (t:ty pe nv ek k) (pf:PolyFn np) {struct t} : 
\forall (ea:enva np nv ek), envf nv ek (ptype pf) ea \rightarrow specType' t (ptype pf) ea :=

match t in type nv ek k
return \forall (ea:enva np nv ek), envf nv ek (ptype pf) ea \rightarrow specType' t (ptype pf) ea with
| tconst nv ek k tc
⇒ fun ea ef \rightarrow match tc return specType' (tconst nv ek tc) (ptype pf) ea with
| tc_unit ⇒ weakening_tconst (punit pf)
| tc_int ⇒ weakening_tconst (pint pf)
| tc_prod ⇒ weakening_tconst (pprod pf)
| tc_sum ⇒ weakening_tconst (psum pf)
end
| tvar nv ek i
⇒ fun ea ef \rightarrow ith_fin (ggetS i ef)
| tapp nv ek k1 k2 t1 t2
⇒ fun ea ef \rightarrow convert_tapp ((instantiate_tuple (replace_fvs t2 ea) (specTerm' t1 pf e a ef)) (specTerm' t2 pf ea ef))
| tlam nv ek k1 k2 t'
⇒ fun ea ef \rightarrow convert_tlam
(dep_curry
 (fun tup ⇒ specType' (tvar (S nv) (k1, ek) (fz nv)) (ptype pf) (tup, ea))
 (fun As : tupleT (decK k1) np ⇒
  (fun fa : specType' (tvar (S nv) (k1, ek) (fz nv)) (ptype pf) (As, ea) ⇒
   specTerm' t' pf (As, ea) (weakening_envf (fa, ef)))))
end.

(∗ Special case for closed types ∗)

Definition specTerm (np:nat) (k:kind) (t:closed_type k) (pf:PolyFn np) : specType t (ptype pf) :=
specTerm' t pf tt tt.

Figure 4. Term specialization

Proof. Unfolding definitions (Figure 3), we find that we have to prove that

\((\forall \{T\}1 [U]1, \ldots, [T]_{np} [U]_{np}) = (\forall \{T U\}1, \ldots, [T U]_{np})\)

The equalities between the elements are trivial (replacing free variables before or after application gives the same result), so we can complete the proof by induction on the length of the tuples. □

6.4 Lambda abstraction

To specialize a polytypic function pf of type Pt to a lambda abstraction (ΛA . T) we first construct the term

\[\text{fun } (A_1, \ldots, A_{np}) f_A \Rightarrow \text{specTerm' } T \text{ pf } ((A_1, \ldots, A_{np}), ea) (f_A, ef)\]

which we then curry to get the required term

\[\text{fun } A_1 \ldots A_{np} f_A \Rightarrow \text{specTerm' } T \text{ pf } ((A_1, \ldots, A_{np}), ea) (f_A, ef)\]

Currying this function is, however, not entirely straightforward. The type of the body of this function

\[\text{specType' } T \text{ pf } ((A_1, \ldots, A_{np}), ea)\]

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

Fixpoint dep_curry A n :
\forall (C : tupleT A n → Set)
(f : \forall (x : tupleT A n), C x),
quantify_tuple C :=
match n return \forall (C : tupleT A n → Set)
(f : \forall (x : tupleT A n), C x),
quantify_tuple C with
| 0 ⇒ fun c f ⇒ f tt
| S m ⇒ fun c f a ⇒
  dep_curry A m (fun args ⇒ c (a, args))
  (fun args ⇒ f (a, args))
end.

The result of dep_curry is something of the form quantify_tuple, which we described in Section 5. However, the return type we want is

\[\text{specType' } (\text{tlam } T) \text{ pf } ea\]

we can construct a term of type

\[\text{specType' } (\text{tlam } T) \text{ pf } ea\]

Proof. Unfolding definitions (Figure 3) we find that we have to prove that given a term of type

\[\text{quantify_tuple} (\text{fun } As : \text{tupleT} (\text{decK } k1) \text{ np} ⇒ \text{specType' } (\text{tvar } (S \text{ nv}) (k1, ek) (fz \text{ nv})) \text{ Pt } (\text{As, ea}) ⇒ \text{specType' } T \text{ Pt } (\text{As, ea}))\]

we can construct a term of type

\[\text{specType' } (\text{tlam } T) \text{ pf } ea\]
Applying this lemma leaves us to prove that the two arguments to type tensional equality:

\[ \forall \alpha . T \alpha \Rightarrow \forall \alpha . T' \alpha \]
even if we can prove that \( T \alpha = T' \alpha \) for any \( \alpha \). We can, however, construct a term of type \( \forall \alpha . T' \alpha \) given a term of type \( \forall \alpha . T \alpha \) (or vice versa): we can only prove that these two types are isomorphic. To prove the isomorphism

\[ \text{quantify_tuple } C \cong \text{quantify_tuple } C' \]

We need an auxiliary lemma that \text{quantify_tuple} preserves extensional equality:

**Lemma 4 (quantify_tuple_ext).** For all \( A, n \), given two functions \( f, g : \text{tupleT} A n \to \text{Set} \), if \( f \) and \( g \) are extensionally equal, that is

\[ \forall \text{tup : tupleT } A n, f \text{ tup} = g \text{ tup} \]

then \text{quantify_tuple} \( f \) and \text{quantify_tuple} \( g \) are isomorphic.

**Proof.** By induction on \( n \). □

Applying this lemma leaves us to prove that the two arguments to \text{quantify_tuple} return the same result given the same input, i.e. for any \( \text{tuple } As = (A_1, \ldots, A_{np}) \):

\[
\begin{align*}
\text{kit k1 Pt (replace_fvs (tvar (S nv) (k1, ek) (fz nv)) (As, ea))} & \to \\
\text{kit k2 Pt (replace_fvs T (As, ea))} & = \\
\text{kit k1 Pt As} & \to \\
\text{kit k2 Pt (apply_tupleT (replace_fvs (tlam T) ea) As)} & \\
\end{align*}
\]

First we prove in lemma \text{tvar_tuple} that, given the environment \( ((A_1, \ldots, A_{np}), \text{ea}) \), we have

\[ ([\text{tvar (S nv) (k1, ek) (fz nv)}]_i, \ldots) = (A_1, \ldots) \]

The equality on the individual elements is trivial: we are replacing free variable \( \text{fz} \) for which we always use elements in the first tuple in the given environment, i.e. elements in \((A_1, \ldots, A_{np})\); the proof is therefore by induction on \( np \). This reduces the problem to

\[
\begin{align*}
\text{kit k1 Pt As} & \to \\
\text{kit k2 Pt (apply_tupleT (replace_fvs (tlam T) ea) As)} & \\
\end{align*}
\]

In lemma \text{replace_bound_var} we then show that

\[ ([\text{tlam T}]_i : A_1, \ldots, [\text{tlam T}]_{np} A_{np}) \]

in environment \( \text{ea} \) is equivalent to

\[ ([T]_1, \ldots, [T]_{np}) \]

in environment \( ((A_1, \ldots, A_{np}), \text{ea}) \). Recall that the bound variable in \( \text{tlam T} \) becomes free in \( T \) and will be replaced by \( A_i \) in \([T]_i\). It is then easy to see that replacing all free variables in \( \text{tlam T} \) using \( \text{ea} \) and applying the result to \( A_i \) is the same as replacing all free variables in \( T \) with \( (A_1, \ldots, A_{np}) \) added to the environment.

Therefore this proof can again be done by induction on \( np \), which completes the proof of \text{convert_tlam}. □

Another difficulty in constructing the specialized term for lambda abstraction is in adding the function \( f_A \) to the environment \( ef \). The existing environment \( ef \) has an entry for each free variable in \( \text{tlam T} \), but variable \( i \) in the lambda abstraction will become variable \( \text{fs} i \) in the body \( T \) of the lambda abstraction.

Therefore each function

\[ f_X : \text{specType'} (\text{tvar nv ek i}) \text{ Pt ea} \]

associated with variable \( i \) in the old environment, should have type

\[ f_X : \text{specType'} (\text{tvar (S nv) (k, ek) (fs i)}) \text{ Pt } ((A_1, \ldots, A_{np}), \text{ea}) \]

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

**Lemma 5 (weakening_envf).** For all \( nv k ek i Pt As ea, the type \( \text{specType'} (\text{tvar nv ek i}) \text{ Pt ea} \) is the same type as \( \text{specType'} (\text{tvar (S nv) (k, ek) (fs i)}) \text{ Pt } ((A_1, \ldots, A_{np}), \text{ea}) \).**

**Proof.** Unfolding definitions (Figure 3), we find that we have to prove

\[
([\text{tvar nv ek i}], \ldots) = ([\text{tvar (S nv) (k, ek) (fs i)}], \ldots)
\]

The equalities between the elements are trivial, so we can complete the proof by induction on the length of the tuple. □

The lemma \text{weakening_envf} makes use of this result to ensure that each of the elements in \( ef \) can be converted to the correct type:

**Lemma 6 (weakening_envf).** For all \( nv k ek Pt As ea, the type \( (\text{specType'} (\text{tvar (S nv) (k, ek) (fs nv)}) \text{ Pt } ((A_1, \ldots, A_{np}), \text{ea})) \) is the same type as \( \text{envf (S nv) (k, ek) Pt (As, ea)} \).**

**Proof.** Unfolding the definition of \( \text{envf} \), we find that we have to prove that the type

\[
\text{gtupleS nv } (\text{fun (i:Fin nv) } \Rightarrow \text{specType'} (\text{tvar nv ek i}) \text{ Pt ea}) \text{ (elements_of_fin nv)}
\]

is the same type as

\[
\text{gtupleS nv } (\text{fun (i:Fin (S nv)) } \Rightarrow \text{specType'} (\text{tvar (S nv) (k, ek) i}) \text{ Pt (As, ea)}) \text{ (mapS fs (elements_of_fin nv))}
\]

As it stands, however, this lemma is impossible to prove. We need to do induction on the length of the tuple

\[
\text{mapS fs (elements_of_fin nv)}
\]

but the length of that tuple is \( nv \)—and we need to keep \( nv \) invariant throughout the proof. Instead we prove a stronger property that abstracts away from \( \text{elements_of_fin} \) and prove the lemma over any tuple \( \text{tup} \) of length \( nv \); this decouples the two occurrences of \( nv \). Hence we need to prove that
The goal of our work is to provide an infrastructure in the proof assistant Coq to do proofs over Generic Haskell-style polytypic programs. This paper is an important step towards this goal and provides a formal definition of polytypic programming in Coq.

We have given definitions for records that describe polytypic functions and their types. Programmers who are familiar with Generic Haskell should easily recognize these structures, as they are almost identical to the description of polytypic functions in those systems. Moreover, we have presented the generic view with associated decoders and defined type specialization and term specialization. Since the result of term specialization is a term of the type computed by type specialization, our implementation is a formal proof that term specialization returns terms of the desired type.

Like our view, the view in Generic Haskell does not support recursion on the type level. Instead, recursive types are supported through value recursion. For example, consider the type $\text{list}^\alpha$. Its “structural” representation as a type in the generic view is $\text{list}^\alpha = \alpha \cdot 1 + \alpha \times \text{list} \alpha$

where $\alpha$ denotes the unit type. Note that $\text{list}^\alpha$ is in terms of the ordinary list type: the recursive occurrences of the list datatype are not replaced. We can then define two functions

$$
\text{fromList} : \forall \alpha, \alpha \rightarrow \text{list}^\alpha \\
\text{toList} : \forall \alpha, \text{list}^\alpha \rightarrow \alpha
$$

which translate from a list to its structural representation and back. We can now define $\text{mapList}$ over lists using the polytypic map function as follows:

$$
\text{Fixpoint mapList} (A B : \text{Set}) (f : A \rightarrow B) (xs : \text{list} A) : \text{list} B := \\
\text{toList} (\text{specTerm'} \ tisto \ \text{map} ((\text{list}, (\text{list}, tt)), tt) (\text{mapList}, tt) A f (\text{fromList} xs))
$$

Since “list” is a free variable in the definition of $\text{list}^\alpha$, we need a variant $\text{specTerm'}$ on $\text{specTerm}$ which allows for open types and accepts two environments of type $\text{envr}$ and $\text{envf}$ (Section 6). In particular, $\text{envf}$ must contain a function of type

$$
\forall (A B : \text{Set}) (f : A \rightarrow B), \text{list} A \rightarrow \text{list} B
$$

which it will apply to the recursive occurrences of $\text{list}$. Obviously, this is the very function we are defining, so we pass $\text{mapList}$ itself. Unfortunately, this definition is not accepted by Coq because the recursive call to $\text{mapList}$ is not made to arguments that are obviously structurally smaller—even though they will be. We need to convince Coq that this function terminates. The work by (Abel 2007) might help to solve this problem, but this is future work.

Since we have a formal definition of term specialization, it is theoretically possible to prove properties about polytypic functions using only the infrastructure we describe in this paper. However, the definition of term specialization is sufficiently involved that additional support is essential. Hinze describes one way to prove such properties in (Hinze 2000a,b); it is our intention to formalize his work in Coq. Once that is completed, we can start to investigate which tactics we can add to Coq that will help solve these proofs, so that we can provide a truly usable framework for Generic Haskell programmers for developing proofs over polytypic functions.

---

7 Both (Altenkirch and McBride 2003) and (Norell 2002) define a generic view that supports arbitrary recursion with associated decoders. This is impossible in Coq, which does not support general type-level recursion so that it can guarantee termination.
Acknowledgments

We would like to thank Ralf Hinze, Johan Jeuring and Yves Bertot for their early feedback on the proposal for this research project and the people on the Coq mailing list for their generous assistance. We would also like to thank the anonymous reviewers for their comprehensive reviews and helpful suggestions.

References


