Polytypic Programming in Coq

Wendy Verbruggen * Edsko de Vries Arthur Hughes
Computer Science Department, Trinity College Dublin, Ireland
{verbruwj, 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 F we can *specialize* f to an ordinary function $f\langle T\rangle$ over a datatype T. The type of $f\langle T\rangle$ is the specialization $F\langle T\rangle$ to the kind of T. Term specialization ($f\langle T\rangle$) is defined by induction on the structure of T; type specialization ($F\langle T\rangle$) is defined by induction on the *kind* of T.

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 $\star$).

As a second more interesting example consider the type `tfork`, defined as $\Lambda\alpha\ .\ (\alpha,\alpha)$. 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`, $\Lambda\alpha\cdot\Lambda\beta\cdot(\alpha,\beta)$, 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`, $\Lambda\phi\cdot\Lambda\alpha\cdot\phi\ \alpha$, 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 $\star$ (`tint`) we get the identity function. For first-order kinds such as $\star\to\star$ or $\star\to\star\to\star$ (`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 $(\star\to\star)\to\star\to\star$ (`tapply`) are supported.

---

* Supported by the Irish Research Council for Science, Engineering and Technology

WGP'08, September 20, 2008, Victoria, BC, Canada.

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 `specTerm` T f must be of the form `specType` T F, our implementation is a formal *proof* that the term specialization $f\langle T\rangle$ must have type $F\langle T\rangle$.

The final point is important since it paves the way towards our ultimate goal of providing an infrastructure in Coq to prove properties about Generic Haskell style programs. 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 as 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 comforting 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 $\star$:

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

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 (`typeKindStar`) of type `nary_fn np Set Set`. A term `nary_fn` n A B denotes the type

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

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

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

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

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

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

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

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

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

(We can ask Coq to simplify these types for us, as shown in the introduction.) We can now define the polytypic `map` function as

```
Definition map : PolyFn 2 := 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 D (f a)
  | inr c ⇒ inr B (g c)
  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

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

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

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

The definition of `equal` is

```
Definition equal : PolyFn 1 := polyFn Equal
 (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).
```

3. Coq

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

3.1 Dependent types

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

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

as

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

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

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

```
Inductive Fin : nat → Set :=
| fz : ∀ n, Fin (S n)
| fs : ∀ n, Fin n → Fin (S n).
```

`Inductive` introduces an inductive datatype[1]; the syntax can be compared to the syntax for GADTs in Haskell (Peyton Jones et al. 2006): we specify the kind of the datatype and list the types of the constructors. Since `Fin` is parametrized by a natural number—a term, not a type—it is a so-called *dependent* datatype. It has two constructors: `fz` corresponds to the natural number zero (and is an element of any set of at least one element) and `fs` corresponds to the successor operation on natural numbers. For example, the term

```
fs (fz 3) : Fin 5
```

denotes the second element in the set $\{0, 1, 2, 3, 4\}$. Although `fs` takes two arguments (a natural number and another element of `Fin`), we only specify one since the first n can be inferred from the second. We say that n is an *implicit argument* of `fs`.

Given `Fin` we can write the indexing operator as

```
Fixpoint getT (A:Type) (n:nat) (i:Fin n) {struct i}
    : tupleT A n → A :=
 match i in Fin n return tupleT A n → A with
 | fz m    ⇒ fun tup ⇒ let (x, xs) := tup in x
 | fs m i' ⇒ fun tup ⇒
             let (x, xs) := tup in getT A i' xs
 end.
```

The syntax "`match` e `in` T `return` σ" denotes a dependent pattern match on $e : T$, where the type of each branch has type σ (which may depend on both e and T). Since the index i must be within bounds due to its type, `getT` is a total function. We use {`struct i`} to tell Coq which argument we want to do structural recursion on so that Coq can verify termination.

3.2 Universe inconsistencies

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

$$(A, B, C)$$

constructs the type

$$A \times B \times C$$

We can define this function as

```
Fixpoint gtupleT (n:nat) : tupleT Type n → Type :=
 match n return tupleT Type n → Type with
 | O   ⇒ fun tup ⇒ unit
 | S m ⇒ fun tup ⇒
         let (t, ts) := tup in t * gtupleT m ts
 end.
```

This takes a nested tuple (tuple of tuples) as argument, i.e. we must construct a tuple of type

$$\texttt{tupleT Type}\ n$$

We will get an unpleasant surprise, however, when we attempt to define

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

because Coq will come back with the uninformative error

```
Universe inconsistency
```

[1] Although it may be slightly unusual to define `tupleT` recursively while defining `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; however, 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 $\star$ in Haskell). Unlike in Haskell, however, this hierarchy continues ad infinitum: `Set` has type $\mathtt{Type}_0$, $\mathtt{Type}_0$ has type $\mathtt{Type}_1$, and generally $\mathtt{Type}_i$ has type $\mathtt{Type}_{i+1}$. Moreover, there is a coercion rule that if $T : \mathtt{Type}_i$ then $T : \mathtt{Type}_j$ for any $j \geq i$. This *stratification* of `Type` prevents the encoding of logical paradoxes (e.g., Hurkens 1995).

The user cannot assign 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 `Type`, infers the constraints on these levels, and verifies that there are no inconsistencies. For `tupleT` we have

$$\mathtt{tupleT} : \mathtt{Type}_i \rightarrow \mathtt{nat} \rightarrow \mathtt{Type}_j \;\; (i \leq j)$$

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 $\mathtt{Type}_j$. The constructed type itself must then have type

$$\mathtt{tupleT}\ \mathtt{Type}_j\ n$$

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

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

$$(x, y, z)$$

of elements of some type A, we construct the type

$$fx \times fy \times fz$$

This is implemented as

```
Fixpoint gtupleT (A:Set) (n:nat) (f:A → Type) :
   tupleS A n → Type :=
 match n return tupleS A n → Type with
 | O   ⇒ fun 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:

```
ggetT : ∀ (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 $\mathtt{Type}_0$). The reason for the two different sorts is that Coq supports *program extraction*: Coq can extract all the computational content (that is, keep the programs but strip the proofs) to be exported to OCaml or Haskell for efficient compilation.

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

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

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 (`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-kinded types can be represented. An element

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

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.

```
(* 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 := type 0 tt k.

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

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

(* 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 (tlam (tapp (tvar 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 kinding derivation

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

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

4.2 Decoding kinds

The decoder for kinds is straight-forward, but there is a subtlety with the choice of `Set` as the decoding of kind $\star$. In the specialization of the arrow kind (Section 5), we will construct types of the form

```
(∀ (α:decK star), ...) : decK star
```

Since the bound variable (α) ranges over the very type that is defined, the type of α 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 `Set` instead, enabling the impredicative `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 nv free variables, we must know the decoded types of the free variables in T. Hence, we need an environment et of type `envt` that associates a decoded type T_i with every free variable i in T. Since the type of T_i (its kind, if you prefer) depends on the 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 generalized product described in Section 3.

We have already introduced two different environments (ek : `envk` nv and et : `envt` nv ek) and we will need two more in the remainder of 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 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 environment.

Figure 1 also shows the encoding for two of the example types that we considered in the introduction: `tfork` and `tapply`. The result of decoding these types is

```
decT tfork tt
 = fun (arg:Set) ⇒ arg * arg
 : decK (karr star star)

decT tapply tt
 = fun (f:Set → Set) (a:Set) ⇒ f a
 : decK (karr (karr star star) (karr star star))
```

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

5. Type specialization

As we saw in Section 2, a polytypic function `map` has a polytypic type `Map`, and the specialization `specTerm` T `map` of `map` to a type T has the specialized type `specType` T `Map`. In this section we explain how to define

```
specType : ∀ (np:nat) (k:kind),
  closed_type k → PolyType np → Set
```

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 $\texttt{Poly}\langle k\rangle$ for some polytypic type `Poly` and defines it by induction on k:

$$\begin{array}{l} \texttt{Poly}\langle k :: \Box\rangle \;::\; k \to \cdots \to k \to \star \\ \texttt{Poly}\langle \star\rangle \;\; T_1 \ldots T_{np} = \textit{(user defined)} \\ \texttt{Poly}\langle k_1 \to k_2\rangle \;\; T_1 \ldots T_{np} = \forall A_1 \ldots A_{np} \,. \\ \quad \texttt{Poly}\langle k_1\rangle \;\; A_1 \ldots A_{np} \to \texttt{Poly}\langle k_2\rangle \;\; (T_1\ A_1) \ldots (T_{np}\ A_{np}) \end{array}$$

The case for kind $\star$ is the user-defined type (`PolyType`, Section 2). We can rewrite the case for arrow kinds as

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

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

$$\begin{array}{l} \Lambda(T_1, \ldots, T_{np}) \,.\, \forall A_1 \ldots A_{np} \,. \\ \texttt{Poly}\langle k_1\rangle \; (A_1, \ldots, A_{np}) \to \texttt{Poly}\langle k_2\rangle \; (T_1\ A_1, \ldots, T_{np}\ A_{np}) \end{array}$$

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

$$\begin{array}{l} \Lambda(T_1, \ldots, T_{np}) \,.\, \Lambda(A_1, \ldots, A_{np}) \,. \\ \texttt{Poly}\langle k_1\rangle \; (A_1, \ldots, A_{np}) \to \texttt{Poly}\langle k_2\rangle \; (T_1\ A_1, \ldots, T_{np}\ A_{np}) \end{array}$$

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

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

to the universally quantified type

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

This function can be implemented as follows:

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

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

$$\texttt{poly}\langle T :: k\rangle :: \texttt{Poly}\langle k\rangle \; (\lfloor T\rfloor_1, \ldots, \lfloor T\rfloor_{np})$$

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

```
(* Every type has an associated kind environment which assigns a kind to each free variable in the type *)
Definition envk (n:nat) : Set := tupleS kind n.

(* Type environment used by the decoder for the translation of free variables. *)
Definition envt (nv:nat) (ek:envk nv) := gtupleT decK ek.

(* Environment of the form ((a1 .. anp), (b1 .. bnp), ..) to keep track of free variable replacements. *)
Definition enva (np nv:nat) (ek:envk nv) := gtupleT (fun k ⇒ tupleT (decK k) np) ek.

(* Environment for functions associated with free variables *)
Definition envf (np nv:nat) (ek:envk nv) (Pt:PolyType np) (ea:enva np nv ek) :=
  gtupleS (fun i ⇒ specType' (tvar nv ek i) Pt ea) (elements_of_fin nv).
```

Figure 2. Overview of Environments

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

(* Apply kit k Pt to the tuple (⌊t⌋1, ..., ⌊t⌋np). *)
Definition specType' (np nv:nat) (k:kind) (ek:envk nv) (t:type nv ek k) (Pt:PolyType np)
    (ea:enva np nv ek) : decK star :=
  kit k Pt (replace_fvs t ea).

(* Specialization of specType for closed types. *)
Definition specType (np:nat) (k:kind) (t:closed_type k) (Pt:PolyType np) : decK star := specType' t Pt tt.
```

Figure 3. Type specialization

The definition of the floor operator $\lfloor\rfloor_i$ 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 `map` performs at the elements of a structure, is $\Lambda A\ B\ .\ A \to B$. When we specialize `map` to a specific datatype, we will need an instance of this operation for each of the arguments of that datatype. Hence if the datatype has nv parameters, we will need nv copies of this operation, each of which will need np type arguments. To keep track of all of these types, we construct an environment ea : `enva` of the form

$$\underbrace{((A_1, \ldots, A_{np}), (B_1, \ldots, B_{np}), (C_1, \ldots, C_{np}), \ldots)}_{nv}$$

The floor operation $\lfloor T' \rfloor_i$ replaces each free variable in T' (each argument of the datatype) by the i'th variable associated with it by creating the tuple $(A_i, B_i, C_i, \ldots)$ and then decoding T' using this new tuple as type environment (Section 4.3).

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

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

[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 ($nv = 0$), ea must be empty and $(\lfloor T\rfloor_1, \ldots, \lfloor T\rfloor_{np})$ degenerates to $(T, \ldots, T)$. From a user's perspective (who will *always* specialize polytypic functions to closed types), this means that all np arguments of a polytypic function will be initialized to the *same* type (see also Section 2).

The full definition of type specialization is shown in Figure 3; `kit` calculates 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 is created by `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 `poly` of type `Poly` to a type $T :: k$ as

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

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

```
specTerm : ∀ (np:nat) (k:kind)
             (t:closed_type k) (pf:PolyFn np),
  specType t (ptype pf)
```

Since `specTerm` returns a term of the type computed by `specType`, the definition of `specTerm` is a formal proof that term specialization returns terms of the required type. The definition of term specialization is shown in Figure 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 `specTerm`.

6.1 Constants

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

```
pprod : specType tprod ptype
```

Recall from Figure 1 that `tprod` is syntactic sugar for

```
tconst 0 tt tc_prod
```

As described in Section 4, instances of `type` encode kinding derivations; `tprod` encodes the derivation in the empty environment

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

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

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

Generally, we need a function of type

```
specType' (tconst nv ek tc_prod) ptype ea
```

for some number of free variables nv and associated kind environment ek (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

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

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 `tconst nv ek tc_prod` is the same as the specialized type of `tconst 0 tt tc_prod`, as proven by the following weakening lemma:

LEMMA 1 (`weakening_tconst`). *For all nv, tc, ek, Pt, ea,*

```
specType (tconst 0 tt tc) Pt
```

is the same type as

```
specType' (tconst nv ek tc) Pt ea
```

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

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

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 f_A 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 ef with an entry for each of these functions.

The tricky part is to assign a type `envf` to ef, since each element in ef has a different type. We can compute `envf` using the generalized tuple from Section 3 as follows[5]

```
gtupleS (fun i ⇒ specType' (tvar nv ek i) Pt ea)
        (elements_of_fin nv).
```

The type of the i'th function is the specialized type of the i'th free variable. Thus, we map `specType` over the tuple $(0, 1, \ldots, nv - 1)$ constructed by `elements_of_fin`. Given ef we can simply return the i'th element in ef as the specialized term for variable i.[6] The *construction* of ef will be considered in the case for lambda abstraction (Section 6.4).

6.3 Application

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

$$\begin{array}{l}\texttt{specTerm' T pf ea ef} : \forall\ \texttt{A}_1 \ldots \texttt{A}_{np}, \\ \quad \texttt{kit k1 Pt } (\texttt{A}_1, \ldots, \texttt{A}_{np}) \to \\ \quad\quad \texttt{kit k2 Pt } (\lfloor \texttt{T} \rfloor_1\ \texttt{A}_1, \ldots, \lfloor \texttt{T} \rfloor_{np}\ \texttt{A}_{np})\end{array}$$

We instantiate the type variables $\texttt{A}_1 \ldots \texttt{A}_{np}$ in `specTerm T pf ea ef` to the elements of the tuple $(\lfloor \texttt{U} \rfloor_1, \ldots, \lfloor \texttt{U} \rfloor_{np})$ using

```
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

$$\begin{array}{l}(\texttt{specTerm' T pf ea ef})\ \lfloor \texttt{U} \rfloor_1 \ldots \lfloor \texttt{U} \rfloor_{np} : \\ \quad \texttt{kit k1 Pt } (\lfloor \texttt{U} \rfloor_1, \ldots, \lfloor \texttt{U} \rfloor_{np}) \to \\ \quad\quad \texttt{kit k2 Pt } (\lfloor \texttt{T} \rfloor_1\ \lfloor \texttt{U} \rfloor_1, \ldots, \lfloor \texttt{T} \rfloor_{np}\ \lfloor \texttt{U} \rfloor_{np})\end{array}$$

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

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

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

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

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

LEMMA 2 (`convert_tapp`). *For all np $k1$ $k2$ ea Pt and types $T : k1 \to k2$ and $U : k1$, the type*

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

is the same type as

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

[5] `gtupleS` is a version of `gtupleT` that returns a `Set` rather than a `Type`.

[6] Due to the way we calculate `envf`, we do need one technical lemma (`ith_fin`) that the i'th element of `elements_of_fin` is i.

```
(* Term specialization *)
Fixpoint specTerm' (np nv:nat) (ek:envk nv) (k:kind) (t:type nv ek k) (pf:PolyFn np) {struct t} :
    ∀ (ea:enva np nv ek), envf nv ek (ptype pf) ea → specType' t (ptype pf) ea :=
  match t in type nv ek k
  return ∀ (ea:enva np nv ek), envf nv ek (ptype pf) ea → specType' t (ptype pf) ea with
  | tconst nv ek k tc      ⇒ fun ea ef ⇒
                             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 ⇒ ith_fin (ggetS i ef)
  | tapp nv ek k1 k2 t1 t2 ⇒ fun ea ef ⇒ convert_tapp
       ((instantiate_tuple (replace_fvs t2 ea) (specTerm' t1 pf ea ef)) (specTerm' t2 pf ea ef))
  | tlam nv ek k1 k2 t'     ⇒ fun ea ef ⇒ convert_tlam
       (dep_curry
         (fun tup ⇒ specType' (tvar (S nv) (k1, ek) (fz nv)) (ptype pf) (tup, ea)
                    → specType' t' (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

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

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 ($\Lambda A\ .\ T$) we first construct the term

```
fun (A₁, ..., A_np) f_A ⇒
  specTerm' T pf ((A₁, ..., A_np), ea) (f_A, ef)
```

which we then curry to get the required term

```
fun A₁ ... A_np f_A ⇒
  specTerm' T pf ((A₁, ..., A_np), ea) (f_A, ef)
```

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

```
specType' T Pt ((A₁, ..., 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
    : ∀ (C : tupleT A n → Set)
        (f : ∀ (x : tupleT A n), C x),
        quantify_tuple C :=
 match n return ∀ (C : tupleT A n → Set)
                  (f : ∀ (x : tupleT A n), C x),
                  quantify_tuple C
 with
 | O   ⇒ fun 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

```
specType' (tlam T) Pt ea
```

We can prove that from a term of the given type we can construct a term of the required type:

LEMMA 3 (`convert_tlam`). *For all $k1$, $k2$, nv, np, Pt, ea and type $T : k_1 \rightarrow k_2$, given a term of type*

```
quantify_tuple (fun As : tupleT (decK k1) np ⇒
 specType' (tvar (S nv) (k1, ek) (fz nv)) Pt (As, ea)
  → specType' T Pt (As, ea))
```

we can construct a term of type

```
specType' (tlam T) Pt ea
```

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

```
quantify_tuple (fun As : tupleT (decK k1) np ⇒
 kit k1 Pt (replace_fvs
  (tvar (S nv) (k1, ek) (fz nv)) (As, ea))
 → kit k2 Pt (replace_fvs T (As, ea)))
```

we can construct a term of type

```
quantify_tuple (fun As : tupleT (decK k1) np ⇒
 kit k1 Pt As → kit k2 Pt
  (apply_tupleT (replace_fvs (tlam T) ea) As))
```

Note that we cannot prove in Coq that these two types are *equal*. Generally, we cannot prove that

$$(\forall\alpha\ .\ T\ \alpha) = (\forall\alpha\ .\ T'\ \alpha)$$

even if we can prove that $T\ \alpha$ and $T'\ \alpha$ are equal for any α. 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

$$\texttt{quantify\_tuple}\ C \cong \texttt{quantify\_tuple}\ C'$$

We need an auxiliary lemma that `quantify_tuple` preserves extensional equality:

LEMMA 4 (`quantify_tuple_ext`). *For all A, n, given two functions f, g :* `tupleT` $A\ n \rightarrow$ `Set`*, if f and g are extensionally equal, that is*

$$\forall\ \texttt{(tup : tupleT}\ A\ n\texttt{)},\ f\ \texttt{tup = }g\ \texttt{tup}$$

then `quantify_tuple` f *and* `quantify_tuple` g *are isomorphic.*

Proof. By induction on n. □

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

```
kit k1 Pt (replace_fvs
       (tvar (S nv) (k1, ek) (fz nv)) (As, ea)) →
 kit k2 Pt (replace_fvs T (As, ea))
=
kit k1 Pt As →
 kit k2 Pt (apply_tupleT
            (replace_fvs (tlam T) ea) As)
```

First we prove in lemma `tvar_tuple` that, given the environment

$$((A_1, \ldots, A_{np}), ea)$$

we have

$$(\lfloor\texttt{tvar (S nv) (k1, ek) (fz nv)}\rfloor_1,\ \ldots) = (\texttt{A}_1,\ \ldots)$$

The equality on the individual elements is trivial: we are replacing free variable `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

```
kit k1 Pt As → kit k2 Pt (replace_fvs T (As, ea))
= kit k1 Pt As →
   kit k2 Pt (apply_tupleT
              (replace_fvs (tlam T) ea) As)
```

In lemma `replace_bound_var` we then show that

$$(\lfloor\texttt{tlam}\ T\rfloor_1\ A_1, \ldots, \lfloor\texttt{tlam}\ T\rfloor_{np}\ A_{np})$$

in environment ea is equivalent to

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

in environment $((A_1, \ldots, A_{np}), ea)$. Recall that the bound variable in `tlam` T becomes free in T and will be replaced by A_i in $\lfloor T\rfloor_i$. It is then easy to see that replacing all free variables in `tlam` T using 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 `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 `tlam` T, but variable i in the lambda abstraction will become variable `fs` i in the body T of the lambda abstraction.

Therefore each function

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

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

$$f_X\ \texttt{: specType' (tvar (S nv) (k, ek) (fs i)) Pt}$$
$$\texttt{((A}_1\texttt{, ..., A}_{np}\texttt{), ea)}$$

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_tvar`). *For all $nv\ k\ ek\ i\ Pt\ As\ ea$, the type*

```
specType' (tvar nv ek i) Pt ea
```

is the same type as

```
specType' (tvar (S nv) (k, ek) (fs i)) Pt (As, ea)
```

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

$$(\lfloor\texttt{tvar nv ek i}\rfloor_1, \ldots) =$$
$$(\lfloor\texttt{tvar (S nv) (k, ek) (fs i)}\rfloor_1, \ldots)$$

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

The lemma `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*

```
(specType' (tvar (S nv) (k, ek) (fz nv)) Pt (As, ea))
  * (envf nv ek Pt ea)
```

is the same type as

```
envf (S nv) (k, ek) Pt (As, ea)
```

Proof. Unfolding the definition of `envf`, we find that we have to prove that the type

```
gtupleS nv
 (fun (i:Fin nv) ⇒ specType' (tvar nv ek i) Pt ea)
 (elements_of_fin nv)
```

is the same type as

```
gtupleS nv
 (fun (i:Fin (S nv)) ⇒
   specType' (tvar (S nv) (k, ek) i) Pt (As, ea))
 (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

$$\texttt{mapS fs (elements\_of\_fin}\ nv\texttt{)}$$

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 `elements_of_fin` and prove the lemma over *any* tuple tup of length m; this decouples the two occurrences of nv. Hence we need to prove that

```
gtupleS m
  (fun (i:Fin nv) ⇒ specType' (tvar nv ek i) Pt ea)
  tup
```

is the same type as

```
gtupleS m
  (fun (i:Fin (S nv)) ⇒
    specType' (tvar (S nv) (k, ek) i) Pt (As, ea))
  (mapS fs tup)
```

This is proven by `weakening_envf_aux` by induction on m. $\square$

7. Related work

There are many different approaches to polytypic programming. In the functional programming community alone these include PolyP (Jansson and Jeuring 1997), Generics for the Masses (Hinze 2006), Derivable Type Classes (Hinze and Jones 2000), Generic Programming, Now! (Hinze and Löh 2006a), Scrap your Boilerplate (Lämmel and Jones 2003, 2005; Hinze and Löh 2006b; Hinze et al. 2006b) and others. A detailed comparison of these approaches is beyond the scope of this paper and we refer to Hinze et al. (2006a) for a thorough survey; none of these approaches, however, support kind-indexed polytypic programming.

In the dependent programming community and the more theoretically oriented type theory community, there are also various proposals for generic programming (Pfeifer and Rueß 1998; Pfeifer and Rueß 1999; Benke et al. 2003; Sheard 2007; Morris et al. 2006, 2007). Most of these proposals use a generic view (universe) construction like we did in this paper; the nature of the generic view dictates to a large extent which generic programs can be written and how they are expressed. None of these approaches uses kind-indexed types, however, and we are specifically interested in proving properties about Generic Haskell programs.

Our work is most closely related to that of Altenkirch and McBride (2003), which gives an implementation of Generic Haskell-style polytypic programming in Oleg (the dependent language developed by Conor McBride in his doctoral thesis, McBride 1999) and Norell (2002), which follows (a preprint of) the first paper closely and presents a similar design in Alfa. The most important difference between our work and theirs is the choice of logic, which must however not be underestimated. Every dependent programming language comes with its own peculiarities, and a solution in one logic will typically not easily transfer to another logic. As mentioned in Section 3.3, an important reason for choosing Coq is the support for proof automation, which is not available in Oleg or Alfa (nor in their successors, Epigram and Agda). This will be extremely beneficial once support for proofs over polytypic programs is in place (see future work). As a second difference, we feel that the transition from Generic Haskell to our library will be easier than the transition to theirs; this is particularly true for (Altenkirch and McBride 2003). Modulo some minor syntactic differences, polytypic functions and their types are defined in our library exactly as they are defined in Generic Haskell, and the specialization of a polytypic function to a datatype is a simple call to `specTerm`. This is important since we want our work to be accessible to Generic Haskell or Generic Clean programmers.

Most of the work on proofs about generic programs comes from the type theory community, rather than from the functional programming community. Consequently, there is very little work about proofs over Generic Haskell-style polytypic programs, other than Hinze's original thesis. One notable exception is (Abel 2007), which is, however, concerned with termination only. Indeed, this is exactly the hole that our work attempts to fill.

8. Conclusions and future work

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 `list`. Its "structural" representation as a type in the generic view is

$$\mathtt{list}^\circ = \Lambda\alpha \cdot 1 + \alpha \times \mathtt{list}\ \alpha$$

where 1 denotes the unit type. Note that $\mathtt{list}^\circ$ is defined in terms of the ordinary list type: the recursive occurrences of the list datatype are not replaced. We can then define two functions

$$\mathtt{fromList} : \forall\alpha, \mathtt{list}\ \alpha \to \mathtt{list}^\circ\alpha$$

and

$$\mathtt{toList} : \forall\alpha, \mathtt{list}^\circ\alpha \to \mathtt{list}\ \alpha$$

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

```
Fixpoint mapList (A B:Set) (f:A → B) (xs:list A)
    : list B :=
  toList (specTerm' tlisto map
       ((list, (list, tt)), tt)
       (mapList, tt) A B f (fromList xs)).
```

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

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

which it will apply to the recursive occurrences of `list`. Obviously, this is the very function we are defining, so we pass `mapList` itself. Unfortunately, this definition is not accepted by Coq because the recursive call to `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[7].

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 write 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 decoder. 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

Andreas Abel. Type-Based Termination of Generic Programs. *Science of Computer Programming*, 2007. MPC'06 special issue. Submitted.

Artem Alimarine. *Generic Functional Programming: Conceptual Design, Implementation and Applications*. PhD thesis, Radbound Universiteit Nijmegen, 2005.

Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 1–20, Deventer, The Netherlands, 2003. Kluwer, B.V. ISBN 1-4020-7374-7.

Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for Generic Programs and Proofs in Dependent Type Theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

Coq Development Team. Coq frequently asked questions (v8.1), 2008. `http://coq.inria.fr/doc-eng.html`.

Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988. ISSN 0890-5401. doi: http://dx.doi.org/10.1016/0890-5401(88)90005-3.

N. G. de Bruijn. A lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

Robert Harper and Robert Pollack. Type Checking with Universes. *Theoretical Computer Science*, 89:107–136, 1991.

Ralf Hinze. Polytypic Values Possess Polykinded Types. In Roland Backhouse and José Nuno Oliveira, editors, *5th International Conference on Mathematics of Program Construction (MPC 2000)*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer–Verlag, 2000a.

Ralf Hinze. Generic Programs and Proofs. Habilitationsschrift, Universität Bonn, Germany, 2000b.

Ralf Hinze. Generics for the Masses. *Journal of Functional Programming*, 16:451–482, 2006.

Ralf Hinze and Simon Peyton Jones. Derivable Type Classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIG–PLAN Haskell Workshop*, Electronic Notes in Theoretical Computer Science, Volume 41.1. Elsevier Science, 2000.

Ralf Hinze and Andres Löh. Generic Programming, Now! In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *School on Datatype-Generic Programming, 2006*, volume 4719 of *Lecture Notes in Computer Science*, pages 150–208. Springer–Verlag, 2006a.

Ralf Hinze and Andres Löh. "Scrap Your Boilerplate" Revolutions. In Tarmo Uustalu, editor, *Mathematics of Program Construction: 8th International Conference (MPC 2006)*, volume 4014 of *Lecture Notes in Computer Science*, pages 180–208. Springer–Verlag, 2006b.

Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing Approaches to Generic Programming in Haskell. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *School on Datatype-Generic Programming, 2006*, volume 4719 of *Lecture Notes in Computer Science*, pages 72–149. Springer–Verlag, 2006a.

Ralf Hinze, Andres Löh, and Bruno Oliveira. "Scrap Your Boilerplate" Reloaded. In *Proceedings of FLOPS 2006*, 2006b.

Antonius J. C. Hurkens. A Simplification of Girard's Paradox. In *TLCA '95: Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, pages 266–278. Springer-Verlag, 1995.

P. Jansson and J. Jeuring. PolyP—A Polytypic Programming Language Extension. In *Conference Record 24th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 470–482. ACM–Press, 1997.

Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03)*, volume 38, pages 26–37. ACM Press, 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate with Class: Extensible Generic Functions. In *Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 204–215. ACM Press, 2005.

Andres Löh. *Exploring Generic Haskell*. PhD thesis, Instituut voor Programmatuurkunde en Algoritmiek, 2004.

Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the Regular Tree Types. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs (TYPES 2004)*, volume 3839 of *Lecture Notes in Computer Science*, pages 252–267. Springer–Verlag, 2006.

Peter Morris, Thorsten Altenkirch, and Neil Ghani. Constructing Strictly Positive Families. In Joachim Gudmundsson and Barry Jay, editors, *Theory of Computing (CATS 2007)*, Conferences in Research and Practice in Information Technology (CRPIT), Volume 65, 2007.

Ulf Norell. Functional generic programming and type theory. Master's thesis, Computing Science, Chalmers University of Technology, 2002.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, 2006.

Holger Pfeifer and Harald Rueß. Polytypic Abstraction in Type Theory. In Roland Backhouse and Tim Sheard, editors, *Workshop on Generic Programming (WGP'98)*. June 1998.

Holger Pfeifer and Harald Rueß. Polytypic proof construction. In *TPHOLs '99: Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pages 55–72, London, UK, 1999. Springer-Verlag. ISBN 3-540-66463-7.

Tim Sheard. Generic Programming in Ωmega. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006.

Wendy Verbruggen. Coq sources for this paper, 2008. `https://www.cs.tcd.ie/~verbruwj/pub/`.