Polytypic Properties and Proofs in Coq

Wendy Verbruggen ∗ Edsko de Vries † Arthur Hughes
Computer Science Department, Trinity College Dublin, Ireland
{verbruvj, devriez, aphughes}@cs.tcd.ie

Abstract
We formalize proofs over Generic Haskell-style polytypic programs in the proof assistant Coq. This makes it possible to do fully formal (machine verified) proofs over polytypic programs with little effort. Moreover, the formalization can be seen as a machine verified proof that polytypic proof specialization is correct with respect to polytypic property specialization.

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

General Terms Languages, Theory, Verification

1. Introduction
In the never ending quest for higher levels of abstraction in programming language research, generic programming has been a hot topic in the functional programming community for a while (Jansson and Jeuring 1997; Hinze and Jones 2000; Lämmel and Visser 2002; Lämmel and Jones 2003; Hinze 2006; Hinze and Löh 2006; Hinze and Löh 2007; Rodríguez et al. 2009). Unfortunately, a consensus on the best approach has yet to be reached, and the number of approaches to generic programming almost equals the number of papers written on the topic. The subject area can be bewildering; survey papers by Hinze et al. (2006) and Rodríguez et al. (2008) try to disentangle some of the various strands of research.

One particular strand that we are interested in is polytypic programming as advocated by Hinze in his seminal habilitationsschrift (Hinze 2000b), which has been incorporated in at least two language designs: Generic Haskell (Löh 2004) and Generic Clean (Alimarine 2005). The goal of our work is to be able to do formal (machine verified) proofs over polytypic programs written in these languages.

A key component of polytypic programming is the specialization of kind-indexed types and the specialization of type-indexed programs. In a companion paper which we published at the Workshop on Generic Programming 2008 (Verbruggen et al. 2008) we demonstrated how type specialization and term specialization can be formalized in the proof assistant Coq (Bertot and Castérane 2004). As well as an important and obvious stepping stone towards formal proofs about such programs, the paper also serves as a formal proof that term specialization is correct with respect to type specialization. We will recap the most important ideas from that paper in Section 2, which will also serve as an introduction to the most important ideas in polytypic programming for readers who may not be familiar with it.

In many ways, the current paper can be seen as the Curry-Howard mirror image of the companion paper. Just like polytypic types are types indexed by a kind, polytypic properties are properties indexed by a kind; and just like polytypic programs are terms indexed by a type, polytypic proofs are proofs indexed by a type. That should come as no surprise, since Curry and Howard tell us that we can read “type” for “property” and “program” for “proof”.

Nevertheless, the structure of properties and proofs (interpreted as types and programs or not) is sufficiently different from the structure of types and programs that it introduces many new difficulties that need to be overcome in order to formalize polytypic proofs.

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

• Although the formal definition of type and term specialization that we have given in the companion paper makes it theoretically possible to do machine verified proofs over polytypic programs, in reality this is almost impossible without further supporting infrastructure. We provide this infrastructure in the current paper, so that formal proofs over polytypic programs can be done with very little effort (we give an example in Section 3.3).

• This paper can be seen as a formal proof that
  • property specialization, the process of specializing a polytypic property to a particular kind, yields well-formed properties (Section 5), and that
  • proof specialization, the process of specializing a polytypic proof to a particular type, is correct with respect to property specialization (Section 6).

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

Some challenges remain; in particular, we do not yet have a fully complete treatment of (co)recursion or (co)induction. However, we have experimented with some approaches and have found two that work. We discuss these in Section 8, and the accompanying Coq scripts contain proofs of concept.

The Coq sources for the formalization described in the paper can be found on the first author’s homepage at http://www.cs.tcd.ie/~verbruvj.
2. Polytypic Functions and their types

This section serves both as an introduction to polytypic programming and as a recap of the companion paper. We will introduce the polytypic (type-indexed) map function, along with its polytypic (kind-indexed) type `Map`. We will demonstrate how `Map` can be specialized to specific kinds and how `map` can be specialized to specific types. We will use the polytypic map function as our running example throughout the paper, and we will see in later sections how to prove the usual functor laws for map, polytypically. For reasons of space, we will have to be brief in this section. For more details, we refer the reader to (Verbruggen et al. 2008) or (Hinze 2006b).

2.1 Type specialization

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

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

The Coq `Record` keyword introduces a (dependent) record of named fields. `PolyType` has one parameter (`np`) and one field (`typeKindStar`) of type `nary_fn np Set Set` (Set is the Coq equivalent of kind `⋆`). A term `nary_fn n = A B` denotes the type

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

We will refer to `np` as the number of type arguments to the polytypic function. We can define the type `Map` of the polytypic map function as a polytypic function of two type arguments:

```coq
Definition Map : PolyType 2 := polyType 2 (fun A B \Rightarrow \map\ A \ B).
```

Type specialization is a two-phase process. First we define the kind-indexed type by induction on the kind `k`. For some polytypic type `Pt` we can informally define this as:

```coq
Pt\ k : k \to \cdots \to k \to \kappa
Pt\ (k_1 \to k_2) \to T_1 \to \cdots \to T_{np} = (\text{fun} np \ \text{defined})
Pt\ (k_1 \to k_2) T_1 \to \cdots \to T_{np} = \forall A_1 \cdots A_{np}.
Pt\ (k_1) A_1 \cdots A_{np} \to Pt\ (k_2) T_1 A_1 \cdots (T_{np} A_{np})
```

`Pt\ k` is a type-level function in `np` arguments; we construct these arguments in the second step. The specialization of a polytypic function `fn` of type `Pt to a type T : k can be defined as:

```coq
SpecType : \forall (np:nat) (k:kind),
closed_type k \to PolyType np \to Set
```

That is, given a closed kind `k` (Section 2.3) and the definition of a polytypic type, we create a “real” Coq type of kind `Set` (which can be read as `⋆`). As an example, the type `Map` of map specialized to `T_example = AA BC`. `A + B × C` is

```
\forall (A1 A2 : Set), (A1 \to A2) →
\forall (B1 B2 : Set), (B1 \to B2) →
\forall (C1 C2 : Set), (C1 \to C2) →
A1 + B1 × C1 → A2 + B2 × C2
```

2.2 Term specialization

A polytypic function is fully specified by giving its polytypic type and the cases for all constants. The terms for all other types can be inferred. Informally, term specialization of a polytypic function `fn` of type `Pt` to a type `T : k can be defined as:

```coq
SpecType : \forall (np:nat) (k:kind),
closed_type k \to PolyType np \to Set
```

That is, given a closed kind `k` (Section 2.3) and the definition of a polytypic type, we create a “real” Coq type of kind `Set` (which can be read as `⋆`). As an example, the type `Map` of map specialized to `T_example = AA BC`. `A + B × C` is

```
\forall (A1 A2 : Set), (A1 \to A2) →
\forall (B1 B2 : Set), (B1 \to B2) →
\forall (C1 C2 : Set), (C1 \to C2) →
A1 + B1 × C1 → A2 + B2 × C2
```

For each free variable `A in T` this definition assumes the existence of a function `fA` which defines what to do with terms of type `A`. Again, such naming conventions cannot be used in a formal development, and we will make use of an environment `e` of the form `(f_1, f_2)` which contains functions for each of the `n` free variables in `T`. In Coq, we define a polytypic function as

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

For the specific case where `ptype` is `Map`, this simplifies to

```coq
punit : unit \to unit
pint : Z \to Z
pprod : \forall (A B:Set), (A \to B) →
  \forall (C D:Set), (C \to D) →
  A \times C \to B \times D
psum : \forall (A B:Set), (A \to B) →
  \forall (C D:Set), (C \to D) →
  A + C \to B + D
```

We can now define the polytypic map function as

```coq
Definition map : PolyFn 2 := polyFn Map
  (fun (u:unit) \Rightarrow u)
  (fun (z:Z) \Rightarrow z)
  (fun (A B:Set) (f:A \to B)
    (C D:Set) (g:C \to D) (x:A \times C) \Rightarrow
    let (a, c) := x in (f a, g c))
  (fun (A B:Set) (f:A \to B)
    (C D:Set) (g:C \to D) (x:A + C) \Rightarrow
    match x with
    | inl a \Rightarrow inl D (f a)
    | inr c \Rightarrow inr (B (g c))
end).
```
This is virtually identical (modulo syntactic differences) to the definition we would provide in Generic Haskell or Generic Clean. Formally, term specialization takes the form:

\[ \text{specTerm} : \forall (np:nat) (k:kind) \text{ (t:closed_type k) (pf:PolyFn np),} \]

\[ \text{specType} t \ (p\text{type pf}) \]

Since \text{specTerm} returns a term of the type computed by \text{specType}, the definition of \text{specTerm} is a formal proof that term specialization returns terms of the required type. Specializing \text{map} to \(T_{\text{example}}\) (above) yields:

\[
\begin{align*}
\text{fun} \ (A1 \ A2 : \text{Set}) \ (f : A1 \to A2) \\
(B1 \ B2 : \text{Set}) \ (g : B1 \to B2) \\
(C1 \ C2 : \text{Set}) \ (h : C1 \to C2) \\
(x : A1 + B1 \times C1) \Rightarrow \\
\text{match} \ x \ \text{with} \\
| \ \text{inl} \ x1 \Rightarrow \text{inl} \ (f \ x1) \\
| \ \text{inr} \ xr \Rightarrow \text{let} \ (xr1, xr2) := xr \ \text{in} \\
\quad \text{inr} \ (g \ xr1, h \ xr2) \\
\end{align*}
\]

2.3 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. Since the result of term specialization should be a function on the “real” Coq datatype, we have to define a mapping from codes in the generic view to ordinary Coq types. Such a mapping is known as a decoder.\(^2\) The definition of the generic view and the types of the decoders are listed in Figure 1.

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.\(^1\) An element

\[ T : \text{type} \ n \ e k \ k \]

is a type of kind \(k\) with at most \(n\) 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 \(env k n\), which is an \(n\)-tuple of kinds.

The definition of the decoder for kinds is straightforward, choosing \(\text{Set}\) as the decoding of kind \(*\). However, for reasons explained in the companion paper, this will require \(\text{Set}\) to be impredicative.

The implementation of the decoder is slightly involved, and we refer the reader to the companion paper for details. Its type can be read as: given a type of kind \(k\) with \(n\) free variables, where the kinds of the free variables are given by \(ek\), we can construct a Coq type of kind \(k\) when given the Coq types of the appropriate kinds for each of the free variables.

3. Polytypic Properties and Proofs

The functor laws for map state that map must preserve identity and composition. The most familiar instance of these laws for functional programmers is the instance for lists, which is usually stated as:

\[
\text{map} \ \text{id} = \text{id} \\
\text{map} \ (f \circ g) = \text{map} \ f \circ \text{map} \ g
\]

However, it is far from evident how to state these properties for an arbitrary datatype \(T\) of arbitrary kind \(k\); much less how to prove them.\(^3\) Fortunately, it turns out that we can state and prove such properties in much the same way as we state the types of polytypic functions and give their implementations. In this section, we will first give a high level description of how polytypic properties can be stated, and then discuss how this can be formalized in Coq. Section 3.3 describes polytypic proofs and finally, Section 3.4 discusses some arguably simpler ways we considered for formalizing polytypic properties, and why none of them were appropriate.

3.1 Stating polytypic properties

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

\[
\text{Id}(k) \ T : \text{map}(k) T T \to \text{Prop}
\]

In the case for kind \(*\) the type \(\text{map}(\*) T T\) specializes to the function type \(T \to T\), and the corresponding definition of the property is:

\[
\text{Id}(\*) T : (T \to T) \to \text{Prop} \\
\text{Id}(\*) T = \lambda f : T \to T . \forall x : T \cdot f \ x = x
\]

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

\[
\forall x : T . \text{map}(T) \ x = x
\]

In other words: in the case for kind \(*\) we have to prove that \(\text{map}(T)\) is itself the identity function.

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

\[
\text{Id}(\* \to *) T : \\
(\forall A1 \ A2 : \* . (A1 \to A2) \to T A1 \to T A2) \to \text{Prop} \\
\text{Id}(\* \to *) T = \\
\lambda f : \forall A1 \ A2 : \* . (A1 \to A2) \to T A1 \to T A2 . \\
\forall A : \* . \text{Id}(\*) A \to \text{Id}(\*) (T A) \\
\equiv \lambda f . \forall A : \* . \lambda g : A \to A . (\forall y : A . g \ y = y) \to \\
\forall x : T A . f \ A \ g \ x = x
\]

Instantiating \(f\) by \(\text{map}(T)\) gives the property familiar from lists:

\[
\text{Id}(\* \to *) T \text{map}(T) = \\
\forall A : \* . \lambda g : A \to A . (\forall y : A . g \ y = y) \to \\
\forall x : T A . \text{map}(T) A \ A \ g \ x = x
\]

Given a type \(A : \*\) and a function \(g : A \to A\) such that \(g\) is the identity function on \(A\), we must show that the property holds for

\(^2\)The requirement for a decoder places restrictions on the universe that we can consider; we will come back to this point in Section 8.

\(^1\)We use De Bruijn indices to represent variables (de Bruijn 1972). The indices in a type of \(n\) free variables are of type \(\text{index} \ n\), which guarantees that no indices can be out of bounds. This is a minor deviation from the definition in the companion paper, where we use a different, but isomorphic, type \(\text{Fin} \ n\). The only difference is that \(\text{Fin}\) is defined inductively, whereas \(\text{index}\) is defined recursively on \(n\); this makes some Coq proofs easier.

\(^3\)As is well-known from category theory, for an arbitrary type \(T\) of fixed first-order kind \(* \to *\) we can state these laws easily, but this does not scale to arbitrary kinds of arbitrary order.
As before, the definition of the property for other kinds can now be formalized:

\[
\text{map}(T) : A \to A. g.
\]

Rephrased, we have to prove that given an identity function \( g, \text{map}(T : \ast \to \ast) g \) is also an identity function.

The property that map preserves composition is more complicated: composition ranges over three functions of type \( \text{Map} \), each instantiated at a different type:

\[
\text{Comp}(k) T_1 T_2 T_3 : \\
\text{Map}(k) T_2 T_3 \times \text{Map}(k) T_1 T_2 \times \text{Map}(k) T_1 T_3 \to \text{Prop}
\]

In the case for kind \( \ast \) the type \( \text{Map}(\ast) \) specializes to the function type \( T_1 \to T_2 \), and the property is defined as:

\[
\text{Comp}(\ast) T_1 T_2 T_3 : \\
(T_2 \to T_3) \times (T_1 \to T_2) \times (T_1 \to T_3) \to \text{Prop}
\]

As before, the definition of the property for other kinds can now be derived. For example, the instance for kind \( \ast \to \ast \) is:

\[
\text{Comp}(\ast \to \ast) T_1 T_2 T_3 : \text{Map}(\ast \to \ast) T_2 T_3 \times \text{Map}(\ast \to \ast) T_1 T_3 \to \text{Prop}
\]

\[
\text{Comp}(\ast \to \ast) T_1 T_2 T_3 = \lambda(f_1, f_2, f_3). \forall A_1 A_2 A_3 : \ast . \\
\forall x : T_1. f_1(f_2 x) = f_3 x
\]

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

### 3.2 Polytypic properties, formally

We define a polytypic property using the following record type:

\[
\text{Record PolyProp (nt nx np:nat) (Pt:PolyType np)} := \\
\text{polyProp \{ \\
\text{idxs : tupleT (tupleT (index nt) np) nx; \\
\text{propKindStar : \forall (types:tupleT (decK star) nt), \\
\text{gtupleTS (kit star Pt)} \to \text{Prop \}; \\
\text{reindex_tuple idxs types) \to Prop \}.}
\]

The record contains two fields: the first (idxs, described in more detail below) gives information about the type of the property, and the second (propKindStar) gives the property for kind \( \ast \).
The record is dependent on four arguments:\(^5\)

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

Given \(nt\) type arguments \(T_1 \ldots T_{nt}\), the type of a polytypic property indexed by a kind \(k\) generally looks like\(^6\):

\[
\text{Pt}(k) \left( T_1, \ldots, T_{nt} \right) \times \cdots \times \text{Pt}(k) \left( T_1, \ldots, T_{nt} \right) \rightarrow \text{Prop}
\]

where \(T_1, \ldots, T_{nt}\) picks the correct \(np\) type arguments for each occurrence of \(\text{Pt}\) from the tuple \(T_1, \ldots, T_{nt}\); e.g., for the case of preservation of composition for map, we have that \(T_1, T_2, T_3\) = \((T_2, T_3)\), \((T_1, T_2, T_3)\) = \((T_1, T_3)\), \((T_1, T_2, T_3)\) = \((T_1, T_3)\);

compare to the type of \(\text{Comp}\), above. This mapping is given by \(\text{idxs}\) in the description of the polytypic property.

The property for kind \(\star\) is given by \(\text{propKindStar}\), given the same tuple \((T_1, \ldots, T_{nt})\) and a tuple

\[
\left( g_1 : \text{Pt}(\star) \left( T_1, \ldots, T_{nt} \right) \right), \ldots, g_{nx} : \text{Pt}(\star) \left( T_1, \ldots, T_{nt} \right) \rightarrow \text{Prop}
\]

Since every element in this second tuple has a different type, the type of the entire tuple is described as a generalized tuple.\(^7\) A generalized tuple \(\text{gtupleTS}\ f \left( x_1, \ldots, x_n \right)\) is a tuple of type \(\left( f \left( x_1 \times \cdots \times f \left( x_n \right) \right) \right)\). In this case, the function \(f\) that we apply is \(\text{kst}\star\ \text{Pt}\), which is the Coq equivalent of \(\text{Pt}(\star)\); and the tuple \(\left( x_1, \ldots, x_n \right)\) that we supply is the tuple of types \((T_1, \ldots, T_{nt})\) \(\rightarrow\) \((T_1, \ldots, T_{nt})\), which is created by \(\text{reindex}_\text{tuple}\).

Hopefully two examples will go a long way towards clarifying these definitions. The property that map preserves identity can be stated using our library in Coq as\(^8\):

\[
\text{Definition Id : PolyProp 1 1 Map :=}
\text{polyProp 1 1 Map :=}
\left( \text{fun F T f \to } \forall x : T, f x = x \right).
\]

Note that we only provide three arguments to \(\text{PolyProp}:\ nt,\ nx\) and \(\text{Pt}\), the argument \(\text{np}\) is implicit in the type of \(\text{Map}\).

\[
\text{Definition Comp : PolyProp 3 3 Map :=}
\text{polyProp 3 3 Map :=}
\left( \text{fun } (T_1, T_2, T_3) (f_1, f_2, f_3) \Rightarrow \forall x : T_1, f_1 (f_2 x) = f_3 x \right).
\]

\[\text{3.3 Polytypic Proofs}\]

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

The definition of a polytypic proof mirrors the definition of a polytypic function (Section 2.2):

\[
\text{Record PolyProof (nt nx np:nat) (pf:PolyFn np) := polyProof { prop : PolyProp nt nx (ptype pf) ; prfUnit : specProp unit prop ; prfInt : specProp int prop ; prfProd : specProp prod prop ; prfSum : specProp sum prop } (\text{closed} \text{prop nt nx np (idxs prop))} ;}
\]

\[
\text{Defined.}
\]

Same as for PolyProp, the argument \(\text{np}\) to PolyProof is implicit in the type of \(\text{map}\) and can therefore be omitted. The details of the proof will be obscure to people not familiar with Coq, but they do not matter for our current purposes. Suffice to say that the proof is easy; the cases for unit and int are solved automatically (by the \text{auto tactic}), and the other cases follow straightforwardly from the appropriate assumptions about the components of the pair or the value in the sum respectively. It is probably possible to write a Coq tactic (proof search algorithm) to prove many of these polytypic proofs fully automatically, but we have left this to future work.

To anticipate the development of proof specialization in Section 6, we can now prove that map specializes to \(\text{texample}\) preserves composition simply by applying proof specialization to the Lemma \(\text{map Comp}\):

\[
\text{specProof texample map Comp}
\]

\[\text{3.4 Alternative definitions}\]

To specify a property using our formalization, the user must specify the type of the property by means of the \(\text{idxs}\) tuple of tuples of indices, and the property for kind \(\star\). The mechanism for specifying the type of the property may seem non-obvious. In this section, we give the rationale for choosing this approach; it can safely be skipped should the reader wish to.

In the definition of a polytypic type (PolyType, Section 2.1) we do not ask the user to specify the kind of the type. We do not need to, because we can construct it given \(\text{np}\); it will always be
equal but not syntactically equal types $T_1$ and $T_2$. Heterogeneous or John Major equality (McBride 2002) is a generalization of the standard equality relation which allows us to state equalities between terms of a different type, even though its only constructor still only allows us to prove equality between terms of the same type:

$$(e : T) \simeq_{T,T} (e : T) \quad \text{JM-REFL}$$

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

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

$$(P_{T_1} e_1 \to P_{T_2} e_2) \
\{\text{ generalize over the proof that } e_1 \simeq_{T_1,T_2} e_2 \} = \forall (pf : e_1 \simeq_{T_1,T_2} e_2). P_{T_1} e_1 \to P_{T_2} e_2 \\{\text{ generalize over } e_1 \} = \forall (x : T_1). (pf : x \simeq_{T_1,T_2} e_2), P_{T_1} x \to P_{T_2} e_2 \\{\text{ replace } T_1 \text{ by } T_2 \} = \forall (x : T_2). (pf : x \simeq_{T_1,T_2} e_2), P_{T_2} x \to P_{T_2} e_2$$

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

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

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

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

$$\text{convert} : \forall A B : \text{Set}, A \equiv B \to A \to B$$

with associated lemma

**LEMMA 1** (Convert Identity).

$$\forall A B (x : A), A = B \to x \equiv_{A,B} \text{convert} x$$

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

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

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

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

When the user defines a polytypic function, they must give the definition of the function for each type constant \( C \), which will have type \( \text{Pt}(k) (\{\emptyset \vdash C : k\}_1, \ldots, \{\emptyset \vdash C : k\}_n) \). Term specialization however is defined over open types, that is, over kind derivations of the form \( \Gamma \vdash T : k \) for some type \( T \) and kind \( k \).

This is important, because even though the user may only apply term specialization to closed types, term specialization is defined by induction on types; when it encounters an abstraction, it needs to introduce a new type assumption into the environment and the body of the lambda is no longer closed.\(^7\) In the companion paper, we therefore proved that

**Lemma 2.**

\[
\text{Pt}(k) (\{\emptyset \vdash C : k\}_1, \ldots, \{\emptyset \vdash C : k\}_n) = \text{Pt}(k) (\{\Gamma \vdash C : k\}_1, \ldots, \{\Gamma \vdash C : k\}_n).
\]

We can prove this lemma by showing that both argument tuples are the same; since type constants contain no free variables, both tuples evaluate to \( (C'_1, \ldots, C'_n) \) where \( C'_i \) is the Coq type that corresponds to \( C \) (the decoding of \( C \)). The specialization of a polytypic function for a type constant is then the definition given by the user converted using the Lemma 2:

**convert (Lemma 2) (user definition)**

During proof specialization we have to prove a similar conversion: we construct a proof of a property \( \text{Pp} \) for some polytypic function \( \text{pfn} \), we have to show that

**Lemma 3.**

\[
\text{Pp}(k) (\{\Gamma \vdash C : k\}_1, \ldots) = \text{Pp}(k) (\{\Gamma \vdash C : k\}_1, \ldots, \{\Gamma \vdash C : k\}_n).
\]

To prove this lemma, we again show that the two argument tuples are the same. We already proved this for the first tuple; remains to show that the second argument tuple are identical. Since terms of the form \( \text{pfn}(\emptyset \vdash C : k) \) have type \( \text{Pt}(k) (\{\emptyset \vdash C : k\}_1, \ldots, \{\emptyset \vdash C : k\}_n) \) but terms of the form \( \text{pfn}(\Gamma \vdash C : k) \) have type \( \text{Pt}(k) (\{\Gamma \vdash C : k\}_1, \ldots, \{\Gamma \vdash C : k\}_n) \), we will need to use heterogeneous equality:

**Lemma 4.**

\[
\text{pfn}(\emptyset \vdash C : k) \\
\cong \text{Pt}(k) (\{\emptyset \vdash C : k\}_1, \ldots, \text{Pt}(k) (\{\Gamma \vdash C : k\}_1, \ldots)
\cong \text{Pt}(k) (\{\emptyset \vdash C : k\}_1, \ldots, \text{Pp}(\Gamma \vdash C : k))
\]

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

**convert (Lemma 2 at \( \emptyset \)) (user definition)**

\[
\cong \text{Pt}(k) (\{\emptyset \vdash C : k\}_1, \ldots, \text{Pt}(k) (\{\emptyset \vdash C : k\}_1, \ldots)
\cong \text{Pp}(\Gamma \vdash C : k)
\]

which follows from Lemma 1. We can now prove Lemma 3 using the method that we sketched above: generalize over Lemma 4, rewrite with Lemma 2, and complete the proof.

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

5. Property Specialization

Section 3.2 explains the general form of the type of a polytypic property. For a specific property, the user specifies the type of the property and gives the property for kind \( k \); the case for kind \( k_1 \rightarrow k_2 \) can then be derived. The informal definition of property specialization is very similar to that of type specialization (Section 2.1):

\[
\text{Pp}(k) T_1 \ldots T_{nt} : \text{Pt}(k) (T_1, \ldots, T_{nt}) \rightarrow \text{Prop}
\]

**Pp(\ast) T_1 \ldots T_{nt} = (user defined)**

\[
\text{Pp}(k_1 \rightarrow k_2) T_1 \ldots T_{nt} =
\lambda(f_1, \ldots, f_{nx}) . \forall A_1 \ldots A_{nt} : k_1 . \forall(g_1, \ldots, g_{nx}) .
\text{Pp}(k_1) (A_1, \ldots, A_{nt}) (g_1, \ldots, g_{nx}) \rightarrow
\text{Pp}(k_2) (T_1 A_1, \ldots, T_{nt} A_{nt})
\]

\(\text{app_f$s$(f_1, \ldots, f_{nx}), g_1, \ldots, g_{nx})\)\)

The Coq formalization of this definition is given as \( \text{kip} \) (kind-indexed property) in Figure 2.

When we compare this definition to the definition of type specialization (Section 2.1), we see that the only significant difference (other than its type) is that the kind-indexed property takes an extra tuple of function arguments \( (f_1, \ldots, f_{nx}) \). Consider the property that map preserves composition (Section 3.1). For lists, we can state this property as

\[
\forall g_1, g_2, g_3 : (g_1 \circ g_2) = g_3 \rightarrow \text{map} g_1 \circ \text{map} g_2 = \text{map} g_3
\]

In this case, \( nx = 3 \), \( (f_1, f_2, f_3) \) will all be instantiated to \( \text{map[\text{List}]} \), the tuple \( (g_1, g_2, g_3) \) corresponds to the three functions in the informal statement of the property, and

\(\text{app_f$s$(f_1, \ldots, f_{nx}), g_1, \ldots, g_{nx})\)\)

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

\[
f_i : \text{Pt}(k_1 \rightarrow k_2) (T_1, \ldots, T_{nt})
\]

\[
g_i : \text{Pt}(k_1) (A_1, \ldots, A_{nt})
\]

From Section 2.1 we know that a polytypic type specialized to an arrow kind \( k_1 \rightarrow k_2 \) takes the form

\[
\forall A_1 \ldots A_{nx} : k_1 . \text{Pt}(k_1) (A_1, \ldots, A_{nx}) \rightarrow \ldots
\]

Hence, we first instantiate \( A_1 \ldots A_{nx} \) in \( f_i \) by \( (A_1, \ldots, A_{nt}) \) to get a term of type

\[
\text{Pt}(k_1) (A_1, \ldots, A_{nt}) \rightarrow \text{Pt}(k_2) (T_1 A_1, \ldots, T_{nt} A_{nt})
\]

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

\[
\text{Pt}(k_2) (T_1 A_1, \ldots, T_{nt} A_{nt})
\]

The function \( \text{app_f$s$} \) does exactly this: instantiate \( f_i \) with the appropriate type arguments and then apply it to \( g_i \) (the definition can be found in the Coq sources but is straight-forward).
Fixpoint kip (k : kind) (nt nx np : nat) (Pt : PolyType np) (Pp : PolyProp nt nx Pt) {struct k} :
  ∀ types : tupleT (decK k) nt, gtupleTS (kit k Pt) (reindex_tuple (idxs Pp) types) → Set :=
  match k return
  | star ⇒ fun types fns ⇒ propKindStar Pt types fns
  | karr k1 k2 ⇒ fun types fns ⇒ quantify_tuple (fun types' : tupleT (decK k1) nt ⇒
  ∀ fns' : gtupleTS (kit k1 Pt) (reindex_tuple (idxs Pp) types'),
  kip k1 Pp types' fns' → kip k2 Pp (apply_tupleT types types') (app_fs fns fns'))
end.

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

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

Figure 2. Property Specialization

Given Pp(k), we can now define property specialization as
Pp(k) (|T|₁, . . . , |T|ₙ)
This follows type specialization (Section 2.1) exactly. The corresponding Coq definition is given as specProp’ in Figure 2 (like specType, specProp instantiates specProp’ to closed types).

6. Proof specialization
Informally, proof specialization can be defined as:
prf(T : k) : Pp(k) ([T]₁, . . . , |T|ₙ) (pfns(T)₁, . . . , pfns(T)ₙ)
prf(C : kC) = (user defined)
prf(⟨A : kₐ⟩) = pA
prf(⟨A : T : k₁ → k₂⟩) = λA₁ . . . Aₙt . λpA . prf(T : k₂)
prf(T U : k₂) = (prf(T : k₁ → k₂)) ([U]₁, . . . , |U|ₙ)
(pfn(U)₁, . . . , pfns(U)ₙ) (pfns(U)ₖ₁)
This definition is very similar to the definition of term specialization that we gave in Section 2.2, except that proofs need an additional tuple of arguments (pfns(T)₁, . . . , pfns(T)ₙ) corresponding to the polytypic functions for which we want to prove the property.

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

ets For each of the nt type arguments to the property, this contains a mapping etsi, (1 ≤ i ≤ nt) from the free variables in T to Coq datatypes so that we can define the decoding |T|ᵢ of T. As explained in Section 2.1, each function argument pfns(T)ᵢ (1 ≤ j ≤ nx) requires a similar environment with a mapping for each of its np type arguments; this environment is given by (etsi).

efs As explained in Section 2.2, each function argument pfns(T)ᵢ requires an environment ef containing functions for the free variables in T; ef is a tuple of nx such environments, one for each argument pfns(T)ᵢ.

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

Figure 3 shows the formal statement (specProof') that given an open type t, a polytypic proof prf over a polytypic function pfns, and given the environments ets, efs and ep, we can specialize the proof to t. The proof is by induction on t, as expected. We do not show the full Coq proof here (it can be found in the sources). Instead, we will discuss the individual cases of the proof below.

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

6.1 Constants
The case for constants is given by the user except that, as explained in Section 4, we need a weakening lemma that says
Pp(k) ([∅ ⊢ C : k₀, . . . ] (pfns(∅ ⊢ C : k), . . . )
= Pp(k) ([Γ ⊢ C : k₀, . . . ] (pfns(Γ ⊢ C : k), . . . ).
The proof of this lemma was also given in Section 4.

6.2 Variables
Recall from Section 2.3 that variables in our universe are represented by De Bruijn indices. To construct the proof for a free variable i, we simply look up the i’th element in environment ep. As for term specialization (Verbruggen et al. 2008, Section 6.2), the trickiest part is to define the type of ep. Informally, the i’th element in ep, corresponding to the proof for the i’th variable, has type
Pp(k) ([|i|, . . . , |i|ₙ] (pfns(|i|), . . . , pfns(|i|ₙ))
The formal definition of ep is given in Figure 3. The construction of ep will be considered when we consider type lambdas in Section 6.4.
The second lemma is a little trickier:

\[
\text{Lemma specProof' (nt nx np nv : nat) (k : kind) (ek : envk nv) :}
\]

\[
\text{proof specialization for open types}
\]

\[
\text{Proof specialization for closed types}
\]

\[
\text{Definition specProof (nt nx np : nat) (k : kind) (t : closed_type k) :}
\]

\[
\text{Definition specProp (nt nx np : nat) (k : kind) (t : closed_type k) :}
\]

6.3 Application

For application \((T U)\) we are given the two induction hypothesis for the types \(T\) and \(U\):

\[
\text{IH}_T : \forall (A_1, \ldots, A_{nt}) (g_1, \ldots, g_{nx}) .
\]

\[
Pp(k_1) (A_1, \ldots, A_{nt}) (g_1, \ldots, g_{nx}) \rightarrow
\]

\[
Pp(k_2) ([T_1], \ldots, [T_{nt}]) (\text{app_fs} pfn(T_1), \ldots, pfn(T_{nx}))
\]

\[
\text{IH}_U : Pp(k_1) ([U_1], \ldots, [U_{nt}]) (pfn(U_1), \ldots, pfn(U_{nx}))
\]

and we need to prove:

\[
Pp(k_2) ([T U_1], \ldots, [T U_{nt}]) (pfn(T U_1), \ldots, pfn(T U_{nx}))
\]

If we instantiate \((A_1, \ldots, A_{nt})\) by \(([U_1], \ldots, [U_{nt}])\) and \((g_1, \ldots, g_{nx})\) by \((pfn(U_1), \ldots, pfn(U_{nx}))\) in \(\text{IH}_T\) and then apply this to \(\text{IH}_U\) we get something of type:

\[
Pp(k_2) ([T_1], [U_1], \ldots, [T_{nt}], [U_{nt}]) (\text{app_fs} pfn(T_1), \ldots, pfn(T_{nx})) (pfn(U_1), \ldots, pfn(U_{nx}))
\]

To get the type we actually need we specify two conversion lemmas. The first conversion is fairly straightforward, and its proof can be found in the companion paper:

**Lemma 5 (convert_app specTerm).** For all types \(T\) and \(U\)

\[
([T_1], [U_1], \ldots, [T_{nt}], [U_{nt}]) = ([T_1], \ldots, [T_{nt}])
\]

The second lemma is a little trickier:

**Lemma 6 (convert_app specProof).** For all types \(T\) and \(U\)

\[
(\text{app_fs} pfn(T_1), \ldots, pfn(T_{nx})) ([U_1], \ldots, [U_{nx}])
\]

\[
\Rightarrow (pfn(k_2) (\text{app_fs} pfn(T_1), \ldots, pfn(T_{nx})) ([U_1], \ldots, [U_{nx}]))
\]

\[
(pfn(T U_1), \ldots, pfn(T U_{nx}))
\]

\[
\text{Figure 3. Proof Specialization}
\]

6.4 Lambda abstraction

In the case of a lambda abstraction \(\Lambda A . T\) we get the induction hypothesis for the body of the abstraction:

\[
\text{IH}_T : Pp(k_2) ([T_1], \ldots, [T_{nt}]) (pfn(T_1), \ldots, pfn(T_{nx}))
\]

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

\[
Pp(k_1 \rightarrow k_2) ([\Lambda A . T_1], \ldots, [\Lambda A . T_{nt}]) (pfn(\Lambda A . T_1), \ldots, pfn(\Lambda A . T_{nx}))
\]

We know that the \(Pp(k_1 \rightarrow k_2)\) takes the form

\[
\forall A_1 \ldots A_{nt} (g_1, \ldots, g_{nx}) .
\]

\[
Pp(k_1) (A_1, \ldots, A_{nt}) (g_1, \ldots, g_{nx}) \rightarrow
\]

\[
Pp(k_2) ([\Lambda A . T_1], \ldots, [\Lambda A . T_{nt}]) (pfn(\Lambda A . T_1), \ldots, pfn(\Lambda A . T_{nx})) (g_1, \ldots, g_{nx})
\]

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

- A set of \(nt\) types, given by \(ets\), which is used to define the decoding of \([T_i]\), \(1 \leq i \leq nt\)
- For each of the \(nx\) function arguments to the property, a function that handles occurrences of terms of type \(A\), given by \(eps\)
- A proof of the property at \(A\), given by \(ep\)

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

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

\[
Pp(k) ([i_1], \ldots, [i_{nx}]) (pfn(i_1), \ldots, pfn(i_{nx}))
\]
for each type variable $i$ of kind $k$, where the decoding is interpreted with respect to the original environment $ets$. However, in the body of the lambda abstraction each of these variables is shifted and is now known as $i + 1$; variable 0 refers to the variable bound by the lambda. That means that we need to convert every proof in the original $ep$ environment to a proof of type

$$\text{Pp}(k)\ (\{i + 1\}, \ldots, [i + 1]_{nt}) \ (\text{pfn}(i + 1), \ldots, \text{pfn}(i + 1))_{nt}$$

where the decoding is now interpreted with respect to the extended environment $ets$. This involves proving that\(^\text{10}\)

**Lemma 7.** For each variable $i$ of kind $k$

$$\text{pfn}(i) \simeq \text{pfn}(i + 1)\ (\text{pfn}(i + 1), \ldots, \text{pfn}(i + 1))_{nt}$$

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

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

Once all environments have been extended we need to apply the induction hypothesis $\text{IH}_T$, but first we will need two conversion lemmas to get a proof of the correct type. The first lemma is again a lemma that we have already proven in the companion paper:

**Lemma 8 (convert_tlamspecTerm_aux).** For all types $A_1, \ldots, A_{nt}$ and the type $T$

$$([A : T], A_1, \ldots, [A : T]_{nt} A_{nt}) = ([T], \ldots, [T]_{nt})$$

where each $[T]$ is decoded with $ets$ extended as described above.

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

**Lemma 9 (convert_tlamspecProof).**

$$\text{appfs}(\text{pfn}(A : T), \ldots, \text{pfn}(A : T)_{nt})$$

$$\equiv (\text{pfn}(A_1), \ldots, \text{pfn}(A)_{nt})$$

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

7. Related work

As mentioned in the introduction, different approaches to polytypic ("data-type generic", "type parametric", "shape parametric") programming abound and the literature is vast; we can only give references for further reading here, and highlight the most important differences. We distinguish between two broad categories: approaches proposed by the functional programming community and proposed by the type theory community.

In the first category, we find PolyP and derivatives (Jansson and Jeuring 1997; Rodriguez et al. 2009), Generics for the Masses (Hinze 2006), Derivable Type Classes (Hinze and Jones 2000), Generic Programming, Now! (Hinze and Löh 2006), Scrap your Boilerplate (Lämmel and Jones 2003; Hinze and Löh 2007) and many others. A detailed comparison of these approaches is beyond the scope of this paper; (Hinze et al. 2006) and (Rodriguez et al. 2008) are two survey papers that are good starting points. None of these approaches however are concerned with proofs over generic programs, and none of these approaches support the kind-indexed programs that typify the approach to polytypic programming we are working with.

Some approaches rely on the representation of datatypes as initial algebras; the “origami” programming presented by Gibbons (2006) is a good example. The same approach has been used in the type theory community; an early example is given by Pfeifer and Rueß (1999). They also give an example of a polytypic proof constructed in this fashion, but no proofs about polytypic programs. The class of datatypes captured by such characterizations is often limited; Benke et al. (2003) extend this class and give more examples of proofs (such as reflexivity of polytypic equality), but datatypes are still limited to first-order kinds.

It is a well-known fact in the dependently typed programming community that generic programs can be written by defining a universe with corresponding decoder; a polytypic program is then defined by induction on codes in the universe (as we have done in this paper). The choice of universe decides the range of types that are covered, the range of programs that can be written, and the style of polytypic programs.

Morris et al. (2006) define a universe with an explicit fixed point combinator, but which is carefully defined so that it covers strictly positive types only. Although their universe does not contain type abstraction and type application, the authors show that polymorphic types can be simulated by codes with free variables; thus, the universe covers polymorphic types of first order kinds. The authors give a definition for a polytypic $\text{map}$ function, and prove the two functor laws (“by easy induction”). So it seems that proofs in their universe do not need the sort of infrastructure we define in this paper. On the other hand, the style of programming is very different and a Generic Haskell programmer would not recognize the definition of $\text{map}$; moreover, the definition of the functor laws is not as direct (for example, a special composition operator needs to be defined for composition of morphisms over environments).

Their universe differs from ours in two other interesting ways. First, decoding types requires an environment of codes, rather than an environment of (decoded) types. As a consequence equality can be defined polytypically even over open types (without requiring additional arguments), because even those free types must be representable in the universe and hence the same polytypic equality function can be used to compare them. Second, their universe does not contain the equivalent of our $\text{tvar}$ constructor. Instead, they have two constructors: one for accessing the “first” (most closely bound) variable, and one for weakening the environment. The authors claim that this simplifies proofs; it would be interesting to see if a similar approach can be adopted in our setup.

Various other researchers have suggested universes that in many ways go well beyond our universe. For example, Morris et al. (2007) extend their earlier work to cover dependent datatypes, although this universe is still restricted to first-order kinds. A very different (but equally expressive) sort of universe is the universe of containers; since the universe looks very different, polytypic programs (programs that are parametrized by a container) are also quite different from their counterparts in our style.

\(^{10}\)In the abstraction case for term specialization, we have a similar but simpler problem, where we needed to prove that the two types in this heterogeneous equality are equal.
The approach to polytypic programming we use, characterized by the use of kind-indexed types and properties, is based on that proposed by Hinze (2000a,b) in his habilitationsschrift and which found its way into two mainstream functional programming languages, Generic Haskell (Löh 2004) and Generic Clean (Almaríne 2005). The informal definitions of type, term and property specialization given in this paper are directly from Hinze, although he gives no explicit definition of proof specialization.

This approach has been the subject of various formalizations in type theory; Altenkirch and McBride (2003) implemented it in Oleg, Norell (2002) presents a similar design in Alfa, and Sheard (2007) goes some way towards a design in Omega. None of these formalizations attempt to do any proofs over polytypic programs.

Finally, Abel (2009) gives an alternative formalization of the universe of strictly positive types by annotating the function kind by its variance, and uses sized types to guarantee that induction is well-founded. This work is based closely on Hinze’s, and in particular Abel considers kind-indexed types. However, he does not consider proofs about polytypic programs.

8. Future Work

The careful reader will have noticed that our universe does not contain any notion of recursion. We cannot simply add a general recursion operator, since Coq does not support generic recursion at the type level, we would be unable to define the decoder. As we have seen in the section on related work, it is possible to define a fixed point combinator which is restricted to strictly positive types. However, since our primary goal is to do proofs about Generic Haskell or Generic Clean-style programs, and since the choice of universe determines the style of programs one writes, we wanted our universe to be as close as possible to the universes used in these languages. For instance, the universe suggested by Morris et al. (2006) yields programs that are unrecognizable to Generic Clean programmers. The approach suggested by Abel (2009) is a lot closer, but is restricted to first-order types. Ultimately, the universe of strictly positive types is not large enough.

In fact, the universe used in Generic Clean does not include recursion at all (Almaríne 2005). Instead, recursion is handled at the term level. To define the map function over lists (say), one defines a datatype */List* = *A . 1 + A × List A* (its “structural representation”), which corresponds to the top-level (shallow) deconstruction of a list. Note that */List* is not recursive: the tail is an ordinary list. Obviously, */List* and */List* are isomorphic; given the two witnesses of the isomorphism */toList* and */fromList* (the “bindmap”), we can now define map over lists as follows:

\[
\begin{align*}
\text{mapList} & : (a \to b) \to \text{List } a \to \text{List } b \\
\text{mapList } f & = \\
\text{toList } \cdot \text{map } (\text{List } f ) & \cdot \text{fromList}
\end{align*}
\]

That is, we first decompose the list, then apply the polytypic map function, and finally compose the list again. Since */List* is a free variable in */List*, */map(List)* needs an argument that tells it what to do with objects of type */List*: obviously, this is the very function */mapList* that we are defining.

Although this definition is perfectly adequate in Haskell or Clean, it is rejected by Coq because Coq cannot verify that it terminates. We have experimented with various solutions to this problem, and although we have to leave the details to future work, we have found two approaches that work well (proof of concepts can be found in the accompanying Coq scripts). Both solutions rely on coinduction. This is justifiable, as we are reasoning about Haskell programs; in particular, the list datatype in Haskell describes both finite and infinite lists. Coinductive functions do not have to terminate, but must be productive (speaking informally, they must always be able to produce the next part of the result in finite time). Like termination, productivity is enforced by Coq syntactically: every recursive call must be guarded by a constructor of a coinductive datatype (Bertot and Castérane 2004, Section 13.3).

The simplest way to define */mapList* in a coinductive way is to make use of the partiality monad (Capretta 2005). The partiality monad can be defined as follows in Coq:

\[
\begin{align*}
\text{CoInductive Delay } (A : \text{Set}) : \text{Set} := \\
| \text{Now} & : \forall a : A, \text{Delay } A \\
| \text{Later} & : \forall (d : \text{Delay } A), \text{Delay } A.
\end{align*}
\]

This monad can be thought of as capturing the essence of productivity: the productivity requirement for a function can be satisfied simply by guarding each recursive call of a function with the */Later* constructor. The exciting feature of the partiality monad is that it allows us to define a general fixpoint combinator\(^{12}\), which makes it possible to give a straightforward definition of */mapList*. Moreover, partial functions (such as equality, if we include the function space type constant) can easily be defined. The disadvantage of the use of the partiality monad is that all polytypic functions must now be defined in monadic style. For example, the case for products in map becomes

\[
\begin{align*}
\text{fun} & (A B : \text{Set}) : \text{map } (\text{Delay } B) \\
(C D : \text{Set}) & : \text{map } (\text{Delay } D) \\
(x : A \times C) & \Rightarrow \\
\text{let} & (a, c) := x \text{ in} \\
\text{bindD} & (f a) (\text{fun } b \Rightarrow (f b)) \\
\text{bindD} & (g c) (\text{fun } d \Rightarrow (g d)) \\
\text{returnD} & (b, d))
\end{align*}
\]

Moreover, proofs over functions that are defined using this general fixpoint operator are far from straightforward, although this can probably be alleviated using a good partiality library.

The other solution is to try and define */mapList* directly as a coinductive function:

\[
\begin{align*}
\text{CoFixpoint } \text{mapStream } \ldots & := \\
& \text{toStream } \cdot \text{map } (\text{Stream } f ) \cdot \text{mapStream } \cdot \text{fromStream}.
\end{align*}
\]

(Where */Stream* is a coinductive definition of “lists”.) Although the occurrence of */mapList* here is not obviously guarded, guardedness is checked with respect to various reductions, and this definition almost works: the only use of */mapStream* is when dealing with the tail of the stream, which will always be guarded by the constructor inserted by */toStream*. Unfortunately, Coq’s guardedness checker is not quite clever enough to detect this, and the definition is rejected. However, following a suggestion by Russell O’Connor and Bruno Barras on the Coq mailing list (de Vries 2009), if we change */map(Stream)* to continuation passing style and pass */toStream* as the continuation, then the definition does pass the termination checker.

This appears to be the simplest and most promising solution yet, but since generating CPS style functions involves modifying all of the type, term, property and proof specialization, we have left a detailed exploration of this as future work.

---

\(^{11}\) Although we have not included the function space type constant in our grammar, we can easily add it. This will not affect the formalization of term or proof specialization, but it will affect examples, as not all polytypic functions (such as equality) can be defined over this larger universe.

\(^{12}\) It seems however that in an explicitly typed language such as Coq, we need a kind-indexed family of fixpoint operators, all of which have the same basic functionality but pass different type arguments around. This needs further research.
9. Conclusions

The goal of our work is to be able to give fully formal—machine verified—proofs over Generic Haskell-style programs. In a companion paper (Verbruggen et al. 2008), we gave a formalization of type and term specialization in Coq. This was an important first step, and made it theoretically possible to do formal proofs over polytypic programs. However, in reality such proofs are almost impossible without further infrastructure.

In this paper we provide this infrastructure: we give a formal definition of polytypic properties and polytypic proofs, and formalize property and proof specialization. This does not just make formal proofs over polytypic programs possible, it makes them easy. For example, the proof that the polytypic map function preserves composition is only a few lines, and specializing this proof to particular types is as easy as invoking proof specialization with the desired polytypic proof and the target type as arguments.

This paper can also be interpreted as a fully formal proof that proof specialization is correct with respect to property specialization, and that to do a proof over a polytypic function it indeed suffices to give the proof for the specific instances of the polytypic function for the type constants. This means that even for people that are not interested in fully formal proofs but prefer to do “pencil-and-paper” proofs (as many do), this paper should be interesting as a formal guarantee that pencil-and-paper methods are correct.

References


