Polytypic Properties and Proofs in Coq

Wendy Verbruggen * Edsko de Vries † Arthur Hughes

Computer Science Department, Trinity College Dublin, Ireland

{verbruwj, devriee, 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; Rodriguez et al. 2009). Unfortunately, a consensus on the best approach has yet to be reached, and the number of approaches to generic programming almost equals the number of papers written on the topic. The subject area can be bewildering; survey papers by Hinze et al. (2006) and Rodriguez et al. (2008) try to disentangle some of the various strands of research.

One particular strand that we are interested in is *polytypic programming* as advocated by Hinze in his seminal habilitationsschrift (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 Castéran 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 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/~verbruwj`.

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

† The financial support of SFI is gratefully acknowledged.

WGP'09, August 30, 2009, Edinburgh, Scotland, UK.

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 2000b).

2.1 Type specialization

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 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 $\star$). 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 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:

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

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

$\mathtt{Pt}\langle k\rangle$ is a type-level function in np arguments; we construct these arguments in the second step. The specialization of a polytypic function `pfn` of type `Pt` to a type T has type

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

where $\lfloor T\rfloor_i$ is the type T where each free variable A is replaced by A_i. Of course, such naming conventions are not tenable in a formal development. The companion paper explain this in detail; suffice to say here that decoding $\lfloor T\rfloor_i$ with $1 \le i \le np$ is defined with respect to an environment ets of the form

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

where each subtuple $(A_i, B_i, \ldots)$ contains a Coq datatype for each free variable in T. Then to decode $\lfloor T\rfloor_i$ we choose the ith subtuple from ets and decode T using that mapping[1].

The Coq formalization of type specialization takes the form

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

That is, given a closed type of 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 $\star$). As an example, the type `Map` of `map` specialized to $T_{\text{example}} = \Lambda A\ B\ C\ .\ A + B \times C$ is

```
∀ (A1 A2 : Set), (A1 → A2) →
∀ (B1 B2 : Set), (B1 → B2) →
∀ (C1 C2 : Set), (C1 → 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 `pfn` of type `Pt` to a type $T : k$ can be defined as:

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

For each free variable A in T this definition assumes the existence of a function f_A 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 ef of the form $(f_1, \ldots, f_{nv})$ which contains functions for each of the nv free variables in T. In Coq, 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
}.
```

For the specific case where `ptype` is `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 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).
```

[1] The structure of ets has changed slightly since the previous paper, where it was represented as $((A_1, A_2, \ldots, A_{np}), (B_1, B_2, \ldots, B_{np}), \ldots)$. The new structure makes extracting the correct tuple from the environment straight-forward, but it complicates extending the environment somewhat; however, extraction occurs much more frequently than extension.

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

```
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. Specializing `map` to T_{example} (above) yields

```
fun (A1 A2 : Set) (f : A1 → A2)
    (B1 B2 : Set) (g : B1 → B2)
    (C1 C2 : Set) (h : C1 → C2)
    (x : A1 + B1 × C1) ⇒
  match x with
  | inl xl ⇒ inl (f xl)
  | inr xr ⇒ let (xr1, xr2) := xr in
             inr (g xr1, h xr2)
  end
```

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.[3] An element

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

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

$$ek \vdash T : k$$

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

The definition of the decoder for kinds is straight-forward, choosing `Set` as the decoding of kind $\star$. However, for reasons explained in the companion paper, this will require `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 nv 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.

[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.

[3] We use De Bruijn indices to represent variables (de Bruijn 1972). The indices in a type of nv free variables are of type `index` nv, 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 `Fin` n. The only difference is that `Fin` is defined inductively, whereas `index` is defined recursively on n; this makes some Coq proofs easier.

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

$$map\ id = id$$
$$map\ (f \circ g) = map\ f \circ map\ g$$

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.[4] 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 `Map`; since `Map` is kind-indexed, it follows that the property itself is kind-indexed:

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

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

$$\texttt{Id}\langle\star\rangle\ T : (T \to T) \to \texttt{Prop}$$
$$\texttt{Id}\langle\star\rangle\ T = \lambda f : T \to T\ .\ \forall x : T\ .\ f\ x = x$$

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

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

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

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

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

Instantiating f by $\texttt{map}\langle T\rangle$ gives the property familiar from lists:

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

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

[4] As is well-known from category theory, for an arbitrary type T of fixed first-order kind $\star \to \star$ we can state these laws easily, but this does not scale to arbitrary kinds of arbitrary order.

```
(* 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:index 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

(* Decoders *)
decK : kind → Type
decT : ∀ (nv:nat) (k:kind) (ek:envk nv), type nv ek k → envt nv ek → decK k
```

Figure 1. Recap: Generic View and Decoders

$\texttt{map}\langle T\rangle\ A\ A\ g$. Rephrased, we have to prove that given an identity function g, $\texttt{map}\langle T:\star\to\star\rangle\ g$ is also an identity function.

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

$$\begin{aligned}&\texttt{Comp}\langle k\rangle\ T_1\ T_2\ T_3 :\\ &\quad \texttt{Map}\langle k\rangle\ T_2\ T_3 \times \texttt{Map}\langle k\rangle\ T_1\ T_2 \times \texttt{Map}\langle k\rangle\ T_1\ T_3 \to \texttt{Prop}\end{aligned}$$

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

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

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

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

The property for map will then be

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

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

3.2 Polytypic properties, formally

We define a polytypic property using the following record type:

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

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

The record is dependent on four arguments:[5]

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

Given nt type arguments $T_1 \dots T_{nt}$, the type of a polytypic property indexed by a kind k generally looks like[6]

$$\texttt{Pt}\langle k\rangle\ \underset{\sim\sim\sim\sim}{(T_1,\dots,T_{nt})}_1 \times \cdots \times \texttt{Pt}\langle k\rangle\ \underset{\sim\sim\sim\sim}{(T_1,\dots,T_{nt})}_{nx} \rightarrow \texttt{Prop}$$

where $\underset{\sim\sim\sim\sim}{(T_1,\dots,T_{nt})}_i$ picks the correct np type arguments for each occurrence of Pt from the tuple $(T_1,\dots,T_{nt})$; e.g., for the case of preservation of composition for map, we have that $\underset{\sim\sim\sim\sim}{(T_1,T_2,T_3)}_1 = (T_2,T_3)$, $\underset{\sim\sim\sim\sim}{(T_1,T_2,T_3)}_2 = (T_1,T_2)$ and $\underset{\sim\sim\sim\sim}{(T_1,T_2,T_3)}_3 = (T_1,T_3)$; compare to the type of Comp, above. This mapping is given by idxs in the description of the polytypic property.

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

$$(g_1 : \texttt{Pt}\langle\star\rangle\ \underset{\sim\sim\sim\sim}{(T_1,\dots,T_{nt})}_1, \dots, g_{nx} : \texttt{Pt}\langle\star\rangle\ \underset{\sim\sim\sim\sim}{(T_1,\dots,T_{nt})}_{nx})$$

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 gtupleTS f $(x_1,\dots,x_n)$ is a tuple of type $(f\ \ x_1 \times \cdots \times f\ \ x_n)$. In this case, the function f that we apply is kit star Pt, which is the Coq equivalent of $\texttt{Pt}\langle\star\rangle$; and the tuple $(x_1,\dots,x_n)$ that we supply is the tuple of tuples of types $(\underset{\sim\sim\sim\sim}{(T_1,\dots,T_{nt})}_1, \dots, \underset{\sim\sim\sim\sim}{(T_1,\dots,T_{nt})}_{nx})$, which is created by reindex_tuple.

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

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

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

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

[5] For the examples which we consider in this paper, $nt = nx$. However, there are examples in which this is not true; for instance, see the fusion law in (Hinze 2000b, Section 4.3.2, p. 102). This property and its proof can also be found in the Coq sources that accompany this paper.

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

[7] gtupleTS is a particular variety of a generalized tuple; details can be found in the Coq formalization but are not important here. We discuss generalized tuples in more detail in (Verbruggen et al. 2008, Section 3.2).

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

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):

```
Record PolyProof (nt nx np:nat) (pf:PolyFn np) :=
 polyProof {
  prop : PolyProp nt nx (ptype pf) ;
  prfUnit : specProp tunit prop
             (cst_closed tunit pf (idxs prop)) ;
  prfInt  : specProp tint prop
             (cst_closed tint pf (idxs prop)) ;
  prfProd : specProp tprod prop
             (cst_closed tprod pf (idxs prop)) ;
  prfSum  : specProp tsum prop
             (cst_closed tsum pf (idxs prop))
}
```

(where cst_closed generates the tuple of polytypic functions for which we want to prove the property prop, instantiated at the correct types). That is, we specify the property prop and the proofs for the type constants tunit, tint, tprod and tsum. Here is an example: the proof that map preserves composition.

```
Lemma map_Comp : PolyProof 3 3 map.
Proof.
  apply (polyProof map Comp);
  compute; auto; intros.
  destruct x ; rewrite H ; rewrite H0 ; auto.
  destruct x ; [rewrite H | rewrite H0] ; auto.
Defined.
```

Same as for PolyProp, the argument np to PolyProof is implicit in the type of 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 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 specialized to T_{example} preserves composition simply by applying proof specialization to the Lemma map_Comp:

```
specProof T_example map_Comp
```

3.4 Alternative definitions

To specify a property using our formalization, the user must specify the type of the property by means of the idxs tuple of tuples of indices, and the property for kind $\star$. The mechanism for specifying the type of the property may seem non-obvious. In this section, we give the rationale for choosing this approach; 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 np: it will always be

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

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

Unfortunately, the situation is not so simple for properties: as mentioned in Section 3.2, the type of a property looks like

$$\mathtt{Pt}\langle k\rangle\ \underbrace{(T_1, \ldots, T_{nt})}_1 \times \cdots \times \mathtt{Pt}\langle k\rangle\ \underbrace{(T_1, \ldots, T_{nt})}_{nx} \to \mathtt{Prop}$$

where the problem is to find the mapping $\underbrace{(T_1, \ldots, T_{nt})}_i$.

The most obvious solution might seem to simply ask the user to provide the complete type of the property, given the tuple $(T_1, \ldots, T_{nt})$. However, this is far too liberal: specialization relies on a particular shape of the type of the property (see Section 5). Intuitively, the more leeway we give to the user, the less we can assume about the type of the property and the more difficult it becomes to derive properties for kinds other than $\star$, much less automate the derivation of the corresponding proofs.

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

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

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

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

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

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

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

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

4. Reasoning about Equality

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

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

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

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

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

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

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

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

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

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

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

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

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

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

with associated lemma

LEMMA 1 (Convert Identity).

$$\forall A\ B\ (x : A), A = B \to x \simeq_{A,B} \mathtt{convert}\ x$$

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

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

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" converts deeper into terms (so that every element of the vector is converted, rather than the entire vector).

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

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

This is important, because even though the user may only apply term specialization to closed types, term specialization is defined by induction on types; when it encounters an abstraction, it needs to introduce a new type assumption into the environment and the body of the lambda is no longer closed.[9] In the companion paper, we therefore proved that

LEMMA 2.

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

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

$$\texttt{convert}\ \text{(Lemma 2)}\ \textit{(user definition)}$$

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

LEMMA 3.

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

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 tuples are identical. Since terms of the form $\texttt{pfn}\langle\emptyset \vdash C : k\rangle$ have type $\texttt{Pt}\langle k\rangle\ (\lfloor\emptyset \vdash C : k\rfloor_1, \ldots, \lfloor\emptyset \vdash C : k\rfloor_{np})$ but terms of the form $\texttt{pfn}\langle\Gamma \vdash C : k\rangle$ have type $\texttt{Pt}\langle k\rangle\ (\lfloor\Gamma \vdash C : k\rfloor_1, \ldots, \lfloor\Gamma \vdash C : k\rfloor_{np})$, we will need to use heterogeneous equality:

LEMMA 4.

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

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

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

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

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 $\star$; the case for kind $k_1 \to k_2$ can then be derived. The informal definition of property specialization is very similar to that of type specialization (Section 2.1):

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

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

$$\forall g_1\ g_2\ g_3\ .\ (g_1 \circ g_2) = g_3 \to \texttt{map}\ g_1 \circ \texttt{map}\ g_2 = \texttt{map}\ g_3$$

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

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

corresponds to the application of 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

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

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

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

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

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

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

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

The function app_fs does exactly this: instantiate f_i with the appropriate type arguments and then apply it to g_i (the definition can be found in the Coq sources 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
    ∀ types : tupleT (decK k) nt, gtupleTS (kit k Pt) (reindex_tuple (idxs Pp) types) → Set with
  | star ⇒ fun types fns ⇒ propKindStar Pp 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 nv : nat) (k : kind) (ek : envk nv) (t : type nv ek k)
    (Pt : PolyType np) (Pp : PolyProp nt nx Pt) (ets : envts nt nv ek)
  : gtupleTS (kit k Pt) (reindex_tuple (idxs Pp) (replace_fvs t ets)) → 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) (reindex_tuple (idxs Pp) (replace_fvs t (ets_tt nt))) → Set :=
  specProp' t Pp (ets_tt nt).
```

Figure 2. Property Specialization

Given $\mathtt{Pp}\langle k\rangle$, we can now define property specialization as

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

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:

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

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 $(\mathtt{pfn}\langle T\rangle_1,\dots,\mathtt{pfn}\langle T\rangle_{nx})$ corresponding to the polytypic functions for which we want to prove the property.

Like the definition of term specialization, this truly is an informal definition: many details are omitted. In particular, since T can be 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 ets_i $(1 \le i \le nt)$ from the free variables in T to Coq datatypes so that we can define the decoding $\lfloor T\rfloor_i$ of T. As explained in Section 2.1, each function argument $\mathtt{pfn}\langle T\rangle_j$ $(1 \le j \le nx)$ requires a similar environment with a mapping for each of its np type arguments; this environment is given by $\underset{\sim\sim}{(ets)}_j$.

efs As explained in Section 2.2, each function argument $\mathtt{pfn}\langle T\rangle_j$ requires an environment ef containing functions for the free variables in T; *efs* is a tuple of nx such environments, one for each argument $\mathtt{pfn}\langle T\rangle_j$.

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

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

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

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

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

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 ith element in ep, corresponding to the proof for the ith variable, has type

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

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.

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

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

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

Figure 3. Proof Specialization

6.3 Application

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

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

and we need to prove:

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

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

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

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

LEMMA 5 (convert_tapp_specTerm). *For all types T and U*

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

The second lemma is a little trickier:

LEMMA 6 (`convert_tapp_specProof`). *For all types T and U*

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

Proof. The proof involves some manipulation of heterogeneous equalities. Note that Lemma 5 proves that the two types involved in the heterogeneous equality in Lemma 6 are equal. □

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:

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

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

$$\begin{aligned}&\texttt{Pp}\langle k_1 \rightarrow k_2\rangle\ (\lfloor \Lambda A\ .\ T\rfloor_1, \ldots, \lfloor \Lambda A\ .\ T\rfloor_{nt})\\ &\quad (\texttt{pfn}\langle \Lambda A\ .\ T\rangle_1, \ldots, \texttt{pfn}\langle \Lambda A\ .\ T\rangle_{nx})\end{aligned}$$

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

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

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

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

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

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

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

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

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

where the decoding is now interpreted with respect to the extended environment ets. This involves proving that[10]

LEMMA 7. *For each variable i of kind k*

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

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

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

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

LEMMA 8 (`convert_tlam_specTerm_aux`). *For all types $A_1 \ldots A_{nt}$ and the type T*

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

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

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

LEMMA 9 (`convert_tlam_specProof`).

$$\begin{aligned}&\mathtt{app\_fs}\ (\mathtt{pfn}\langle \Lambda A\ .\ T\rangle_1, \ldots, \mathtt{pfn}\langle \Lambda A\ .\ T\rangle_{nx})\\ &\quad(\mathtt{pfn}\langle A\rangle_1, \ldots, \mathtt{pfn}\langle A\rangle_{nx})\\ &\simeq_{(\mathtt{Pt}\langle k_1\to k_2\rangle\ (\lfloor \Lambda A.T\rfloor_1\ A_1,\ldots)_1 \times \cdots),(\mathtt{Pt}\langle k_1\to k_2\rangle\ (\lfloor T\rfloor_1,\ldots)_1 \times \cdots)}\\ &(\mathtt{pfn}\langle T\rangle_1, \ldots, \mathtt{pfn}\langle T\rangle_{nx})\end{aligned}$$

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 `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 `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 `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 (Alimarine 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 Ωmega. 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.[11]

In fact, the universe used in Generic Clean does not include recursion at all (Alimarine 2005). Instead, recursion is handled at the term level. To define the map function over lists (say), one defines a datatype $\mathtt{List}^\circ = \Lambda A\ .\ 1 + A \times \mathtt{List}\,A$ (its "structural representation"), which corresponds to the top-level (shallow) deconstruction of a list. Note that $\mathtt{List}^\circ$ is not recursive: the tail is an ordinary list. Obviously, `List` and $\mathtt{List}^\circ$ are isomorphic; given the two witnesses of the isomorphism `toList` and `fromList` (the "bimap") we can now define map over lists as follows:

```
mapList : (a → b) → List a → List b
mapList f =
  toList · map⟨List°⟩ f mapList · fromList
```

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 $\mathtt{List}^\circ$, $\mathtt{map}\langle\mathtt{List}^\circ\rangle$ 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).

[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.

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 Castéran 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:

```
CoInductive Delay (A : Set) : Set :=
  | Now : ∀ a : A, Delay A
  | Later : ∀ (d : Delay A), Delay A.
```

This monad can be thought of as capturing the essence of productivity: the productivity requirement for a function can be satisfied simply by guarding each 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

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

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:

```
CoFixpoint mapStream ... :=
 toStream · map⟨Stream°⟩ f mapStream · fromStream.
```

(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 $\mathtt{map}\langle\mathtt{Stream}^\circ\rangle$ 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.

[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.

In Generic Haskell or Generic Clean, the compiler can automatically generate the structural representation of a type, the corresponding bimap, and definitions such as `mapStream`, above. In Coq we could implement this through tactics, but this is unsatisfactory: although each definition generated by the tactics can be verified by the Coq proof verifier, we cannot give a formal proof that the *tactic* is correct. We leave the formalization of this aspect as an open question for now.

Finally, we have not covered extensions such as type-indexed datatypes (Hinze et al. 2004) which are used in Generic Haskell.

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

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

Artem Alimarine. *Generic Functional Programming: Conceptual Design, Implementation and Applications*. PhD thesis, Radboud Universiteit Nijmegen, The Netherlands, 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. Kluwer, B.V., 2003.

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. *Coq'Art: Interactive Theorem Proving and Program Development*. Springer–Verlag, 2004.

Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18, 2005.

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

Edsko de Vries. Coq-club archives: Question about guardedness in cofixpoint, April 2009. `http://logical.saclay.inria.fr/-coq-puma/messages/aa72f538ea88b2e7`.

Jeremy Gibbons. Datatype-generic programming. In *School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 1–71. Springer–Verlag, 2006.

Ralf Hinze. Polytypic values possess polykinded types. In *MPC'00: 5th International Conference on Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer–Verlag, 2000a.

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

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

Ralf Hinze and Simon Peyton Jones. Derivable type classes. In *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, Electronic Notes in Theoretical Computer Science, Volume 41.1. Elsevier Science, 2000.

Ralf Hinze and Andres Löh. Generic programming, now! In *School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 150–208. Springer–Verlag, 2006.

Ralf Hinze and Andres Löh. Generic programming in 3D. *Science of Computer Programming*, 2007. To appear.

Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Sci. Comput. Program.*, 51(1-2):117–151, 2004.

Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in Haskell. In *School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 72–149. Springer–Verlag, 2006.

P. Jansson and J. Jeuring. PolyP—a polytypic programming language extension. In *POPL'97: Conference Record 24th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *TLDI'03: ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, volume 38, pages 26–37. ACM Press, 2003.

Ralf Lämmel and Joost Visser. Typed combinators for generic traversal. In *PADL'02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, pages 137–154. Springer–Verlag, 2002.

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

Conor McBride. Elimination with a motive. In *TYPES'00: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 197–216. Springer–Verlag, 2002.

Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In *TYPES'04: Types for Proofs and Programs*, volume 3839 of *Lecture Notes in Computer Science*, pages 252–267. Springer–Verlag, 2006.

Peter Morris, Thorsten Altenkirch, and Neil Ghani. Constructing strictly positive families. In *CATS'07: Theory of Computing*, volume 65 of *Conferences in Research and Practice in Information Technology (CRPIT)*, 2007.

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

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

Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell'08: Proceedings of the First ACM SIGPLAN Symposium on Haskell*, pages 111–122. ACM Press, 2008.

Alexey Rodriguez, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP'09: Proceeding of the 14th ACM SIGPLAN international conference on Functional programming*. ACM Press, 2009. To appear.

Tim Sheard. Generic programming in Ωmega. In *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer–Verlag, 2007.

Wendy Verbruggen, Edsko de Vries, and Arthur Hughes. Polytypic programming in Coq. In *WGP'08: Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, pages 49–60. ACM Press, 2008.