Slightly expanded version of a paper presented at the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99).
Springer LNCS, Vol. 1690, pp. 55 – 72, Sept. 1999.

Polytypic Proof Construction

Holger Pfeifer[1] and Harald Rueß[2]

[1] Universität Ulm
Fakultät für Informatik
D-89069 Ulm, Germany
pfeifer@informatik.uni-ulm.de

[2] SRI International
Computer Science Laboratory
333 Ravenswood Ave.
Menlo Park, CA, 94025, USA
ruess@csl.sri.com

Abstract. This paper deals with formalizations and verifications in type theory that are abstracted with respect to a class of datatypes; i.e *polytypic* constructions. The main advantage of these developments are that they can not only be used to define functions in a generic way but also to formally state polytypic theorems and to synthesize polytypic proof objects in a formal way. This opens the door to mechanically proving many useful facts about large classes of datatypes *once and for all*.

1 Introduction

It is a major challenge to design libraries for theorem proving systems that are both sufficiently complete and relatively easy to use in a wide range of applications (see e.g. [6, 26]). A library for abstract datatypes, in particular, is an essential component of every proof development system. The libraries of the Coq [1] and the Lego [13] system, for example, include a number of functions, theorems, and proofs for common datatypes like natural numbers or polymorphic lists. In these systems, myriads of mostly trivial developments are carried out separately for each datatype under consideration. This increases the bulk of proving effort, reduces clarity, and complicates lemma selection. In contrast, systems like Pvs [20] or Isabelle [22] support an infinite number of datatypes by using meta-level functions to generate many standard developments from datatype definitions. Pvs simply uses an extension of its implementation to generate axiomatized theories for datatypes including recursors and induction rules, while Isabelle's datatype extension includes tactics to prove these theorems. Both the Pvs and the Isabelle approach usually work well in practice. On the other hand, meta-level functions must be executed separately for each datatype under consideration, and construction principles for proofs and programs are operationalized and hidden in meta-level functions which are encoded in the implementation language of the proof system.

In this paper we propose techniques for building up library developments from a core system without resorting to an external meta level. Moreover, proof objects are constructed explicitly and developments from the library can be used by simply instantiating higher-order quantifiers. Hereby, we rely on the concept of *polytypic abstraction*.

The *map* functional on the list datatype L illustrates the concept of polytypic abstraction exceptionally well. Applying this functional to a function f and a source list l yields a target list obtained by replacing each element a of l with the value of $f(a)$, thereby preserving the structure of l. The type of *map* in a Hindley-Milner type system, as employed by current functional programming languages, is abstracted with respect to two type variables A and B:

$$map : \ \forall A, B.\ (A \to B) \to L(A) \to L(B)$$

Thus, *map* is a *polymorphic* function. The general idea of the *map* function of transforming elements while leaving the overall structure untouched, however, is not restricted to lists and applies equally well to other datatypes. This observation gives rise to a new notion of polymorphism, viz. *polytypy*,[1] for defining functions uniformly on a class of (parameterized) datatypes $\mathcal{T}$:

$$map : \ \forall \mathcal{T}.\ \forall A, B.\ (A \to B) \to \mathcal{T}(A) \to \mathcal{T}(B)$$

Notice that the notion of polytypy is completely orthogonal to the concept of polymorphism, since every "instance" of the family of polytypic *map*-functions is polymorphic. Many interesting polytypic functions have been identified and described in the literature [10, 11, 15–17, 25, 27], and concepts from category theory have proven especially suitable for expressing polytypic functions and reasoning about them. In this approach, datatypes are modeled as initial objects in categories of functor-algebras [14], and polytypic constructions are formulated using initiality without reference to the underlying structure of datatypes.

The concept of polytypy, however, is not restricted to the definition of polytypic functions solely, but applies equally well to other entities of the program and proof development process like specifications, theorems, or proofs. Consider, for example, the *no confusion* theorem. This theorem states that terms built up from different constructors are different. It is clearly polytypic, since it applies to all initial datatypes.

In the following, we examine techniques for expressing polytypic abstraction in type theory. These developments can not only be used to polytypically define functions but also to formally state polytypic theorems and to interactively develop polytypic proofs using existing proof editors. Thus, formalization of polytypic abstraction in type theory opens the door to proving many useful facts about large classes of datatypes *once and for all* without resorting to an external meta-language.

[1] Sheard [25] calls these algorithms *type parametric*, Meertens [15] calls them *generic*, and Jay and Cockett [9] refer to this concept as *shape polymorphism*.

The paper is structured as follows. The formal setting of type theory is sketched in Section 2, while Section 3 includes type-theoretic formalizations of some basic notions from category theory that are needed to specify, in a uniform way, datatypes as initial objects in categories of functor algebras. Furthermore, Section 3 contains generalizations of the usual reflection and fusion theorems for recursors on initial datatypes—as stated, for example, in [2]—to recursors of dependent type, which correspond to structural induction schemes. These developments are polytypically abstracted for the *semantically* characterized class of initial datatypes. This notion of semantic polytypy, however, is not appropriate in many cases where inspection of the form of the definition of a datatype is required. Consequently, in Section 4, we develop the machinery for abstracting constructions over a certain *syntactically* specified class of datatypes. Since the focus is on the generation of polytypic proofs rather than on a general formalization of inductive datatypes in type-theory we restrict ourselves to the class of parameterized, polynomial (sum-of-products) datatypes in order to keep the technical overhead low. The main idea is to use representations that make the internal structure of datatypes explicit, and to compute type-theoretic specifications for all these datatypes in a uniform way. This approach can be thought of as a simple form of *computational reflection* (e.g. [23,29]). Developments that are abstracted with respect to a syntactically characterized class of datatypes are called *syntactically polytypic* in the following. We demonstrate the expressiveness of syntactic polytypy with a mechanized proof of the bifunctoriality property for the class of polynomial datatypes and a polytypic proof of the *no confusion* theorem. Finally, Section 5 concludes with some remarks.

The constructions presented in this paper have been developed with the help of the LEGO [13] system. For the sake of readability we present typeset and edited versions of the original LEGO terms and we take the freedom to use numerous syntactic conventions such as infix notation and pattern matching.

2 Preliminaries

Our starting point is the *Extended Calculus of Constructions (ECC)* [12] enriched with the usual inductive datatypes. We sketch basic concepts of this type theory, fix the notation, and discuss the treatment of datatypes in type theory. More interestingly, we introduce n-ary (co-)products and define functions on generalized (co-)products by recursing along the structure of the descriptions of these types. This technique has proven to be essential for the encoding of syntactic polytypy in Section 4.

The type constructor $\Pi x : A.\ B(x)$ is interpreted as the collection of dependent functions with domain A and codomain $B(a)$ with a the argument of the function at hand. Whenever variable x does not occur free in $B(x)$, $A \to B$ is used as shorthand for $\Pi x : A.\ B(x)$; as usual, the type constructor $\to$ associates to the right. λ-abstraction is of the form $(\lambda\, x : A.\ M)$ and abstractions like $(\lambda\, x : A, y : B.\ M)$ are shorthand for iterated abstractions $(\lambda\, x : A.\ \lambda\, y : B.\ M)$. Function application associates to the left and is written as $M(N)$, as juxtaposition $M\ N$, or even in subscript notation M_N. Types of the form $\Sigma x : A.\ B(x)$ comprise dependent pairs $(.,.)$, and π^1, π^2 denote the projections on the first and second position, respectively. Sometimes, we decorate projections with subscripts as in $\pi^1_{A,B}$ to indicate the source type $\Sigma x : A.\ B(x)$. Finally, types are collected in yet other types *Prop* and Type_i ($i \in \mathbb{N}$). These universes are closed under the type-forming operations and form a fully cumulative hierarchy [12]. Although essential to the formalization of many programming concepts, universes are tedious to use in practice, for one is required to make specific choices of universe levels. For this reason, we apply—carefully, without introducing inconsistencies—the *typical ambiguity* convention [7] and omit subscripts i of type universes Type_i.

Definitions like $c(x_1 : A_1, \ldots x_n : A_n) : B \quad ::= \quad M$ are used to introduce a name c for the term M that is (iteratively) abstracted with respect to x_1 through x_n. The typing $: B$ is optional and specifies the type of the term M. Consider, for example, the definition of the polymorphic identity function I and (infix) function composition $\circ$.

$$I(A \mid \mathit{Type},\ x : A) :\ A \quad ::= \quad x$$

$$.\circ.(A, B \mid \mathit{Type},\ f : B \to C,\ g : A \to B) :\ A \to C \quad ::= \quad \lambda\, x : A.\ f(g(x))$$

Bindings of the form $x \mid A$ are used to indicate parameters that can be omitted in function application. Systems like LEGO are able to infer the hidden arguments in applications like $f \circ g$ automatically (see [13]).

Using the principle of *propositions-as-types*, the dependent product type $\Pi x : A.\ B(x)$ is interpreted as logical universal-quantification and if $M(a)$ is of type $B(a)$ for all $a : A$ then $\lambda\, x : A.\ M(x)$ is interpreted as a proof term for the formula $\Pi x : A.\ B(x)$. It is possible to encode in *ECC* all the usual logical connectives ($\top$, $\bot$, $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\ldots$) and quantifiers ($\forall$, $\exists$, $\exists^1$, $\ldots$) together with a natural-deduction style calculus for a higher-order constructive logic. A logical formula is said to be valid if and only if it is *inhabited*, i.e. a proof term can be constructed for this formula. Leibniz equality ($=$) identifies terms having the same properties.

$$. = .(A \mid \mathit{Type})(x, y : A) : \mathit{Prop} \quad ::= \quad \Pi P : A \to \mathit{Prop}.\ P(x) \to P(y)$$

This equality is intensional in the sense that $a \ = \ b$ is inhabited in the empty context if and only if a and b are convertible; i.e. they are contained in the least congruence $\simeq$ generated by β-reduction. Constructions in this paper employ, besides Leibniz equality, a (restricted) form of extensional equality (denoted $\doteq$) on functions.

$$.\doteq.(A, B \mid \mathit{Type})(f, g : A \to B) : \mathit{Prop} \quad ::= \quad \forall\, x : A.\ f(x) \ = \ g(x)$$

Inductive datatypes can be encoded in type theories like *ECC* by means of impredicative quantification [3]. For the well-known imperfections of these encodings—such as noninhabitedness of structural induction rules—however, we prefer the introduction of datatypes by means of formation, introduction, elimination, and equality rules [4,18,21]. Consider, for example, the extension of type theory with (inductive) products. The declared constant $. \times .$ forms the product type from any pair of types, and pairing $(.,.)$ is the only constructor for this newly formed product type.

$$
\begin{aligned}
&. \times . : \mathit{Type} \to \mathit{Type} \to \mathit{Type} \\
&(.,.) : \Pi A, B \mid \mathit{Type}.\ A \to B \to (A \times B)
\end{aligned}
$$

The type declarations for the product type constructor and pairing represent the formation and introduction rules of the inductive product type, respectively. These rules determine the form of the elimination and equality rules on products.

$$
\begin{aligned}
\mathit{elim}^{\times} \ :\ & \Pi A, B \mid \mathit{Type},\ C :\ (A \times B) \to \mathit{Type}. \\
& (\Pi a : A, b : B.\ C(a,b)) \to \Pi x : (A \times B).\ C(x)
\end{aligned}
$$

Elimination provides a means to construct proof terms (functions) of propositions (types) of the form $\Pi x : (A \times B).\ C(x)$. The corresponding equality rule is specified in terms of a left-to-right rewrite rule.

$$
\mathit{elim}^{\times}_{C}\ f\ (a,b) \ \rightsquigarrow\ f(a)(b)
$$

It is convenient to specify a *recursor* as the non-dependent variant of elimination in order to define functions such as the first and second projections.

$$
\begin{aligned}
\mathit{rec}^{\times}(A, B, C \mid \mathit{Type}) &\ ::=\ \mathit{elim}^{\times}_{\lambda \_ : A \times B.\ C} \\
\mathit{fst} : (A \times B) \to A &\ ::=\ \mathit{rec}^{\times}(\lambda\, x : A, y : B.\ x) \\
\mathit{snd} : (A \times B) \to B &\ ::=\ \mathit{rec}^{\times}(\lambda\, x : A, y : B.\ y)
\end{aligned}
$$

Moreover, the (overloaded) $. \times .$ functional is a bifunctor (see also Def. 2) and plays a central role in categorical specifications of datatypes; it is defined by means of the *split* functional $\langle f, g \rangle$.

$$
\begin{aligned}
&\langle .,. \rangle (C \mid \mathit{Type})(f : C \to A,\ g : C \to B) :\ C \to (A \times B) \\
&\qquad ::=\ \ \lambda\, x : C.\ (f(x),\ g(x)) \\
&. \times .(A, B, C, D \mid \mathit{Type})(f : A \to C,\ g : B \to D) : (A \times B) \to (C \times D) \\
&\qquad ::=\ \ \langle f \circ \mathit{fst}_{A,B},\ g \circ \mathit{snd}_{A,B} \rangle
\end{aligned}
$$

The specifying rules for coproducts $A + B$ with injections $\mathit{inl}_{A,B}(a)$ and $\mathit{inr}_{A,B}(b)$ are dual to the ones for products. Elimination on coproducts is named elim^{+} and its non-dependent variant $[f, g]$ is pronounced "case f or g".

$$
\begin{aligned}
&[.,.](A, B, C \mid \mathit{Type}) :\ (A \to C) \to (B \to C) \to (A + B) \to C \\
&\qquad ::=\ \ \mathit{elim}^{+}_{\lambda \_ : A+B.\ C}
\end{aligned}
$$

Similar to the case of products, the symbol $+$ is overloaded to also denote the bifunctor on coproducts.

$$
\begin{aligned}
&. + .(A, B, C, D \mid \mathit{Type},\ f : A \to B,\ g : C \to D) : (A + C) \to (B + D) \\
&\qquad ::=\ \ [\mathit{inl}_{B,D} \circ f,\ \mathit{inr}_{B,D} \circ g]
\end{aligned}
$$

Unlike products or coproducts, the datatype of parametric lists with constructors *nil* and $(\,.\, :: \,.\,)$ is an example of a genuinely recursive datatype.

$$\begin{array}{rcl} L & : & \mathit{Type} \to \mathit{Type} \\ \mathit{nil} & : & \Pi A : \mathit{Type}.\ L(A) \\ (\,.\, :: \,.\,) & : & \Pi A \mid \mathit{Type}.\ A \times L(A) \to L(A) \end{array}$$

These declarations correspond to formation and introduction rules and completely determine the form of list elimination,[2]

$$\begin{array}{l} \mathit{elim}^L \ : \ \Pi A \mid \mathit{Type},\ C : L(A) \to \mathit{Type}. \\ \qquad\qquad (C(\mathit{nil}_A) \times (\Pi(a,l) : (A \times L(A)).\ C(l) \to C(\ a :: l\ )) \\ \qquad\qquad\quad \to \Pi\, l : L(A).\ C(l) \end{array}$$

and of the rewrites corresponding to equality rules:

$$\begin{array}{rcl} \mathit{elim}^L_C\ f\ \mathit{nil}_A & \rightsquigarrow & \mathit{fst}(f) \\ \mathit{elim}^L_C\ f\ (\ a :: l\ ) & \rightsquigarrow & \mathit{snd}(f)\ (a,l)\ (\mathit{elim}^L_C\ f\ l) \end{array}$$

The non-dependent variants rec^L and hom^L of list elimination are used to encode structural recursive functions on lists.

$$\begin{array}{l} \mathit{rec}^L(A, C \mid \mathit{Type})(f : C \times ((A \times L(A)) \to C \to C)) \ : \ L(A) \to C \\ \qquad ::= \ \ \mathit{elim}^L_{\lambda\_:L(A).\ C}(f) \end{array}$$

$$\begin{array}{l} \mathit{hom}^L(A, C \mid \mathit{Type})(f : C \times ((A \times C) \to C)) : \ L(A) \to C \\ \qquad ::= \ \ \mathit{rec}^L(\mathit{fst}(f),\ \ \lambda(a,\_) : A \times L(A),\ y : C.\ \mathit{snd}(f)(a,y)) \end{array}$$

The name hom^L stems from the fact that $\mathit{hom}^L(f)$ can be characterized as a (unique) homomorphism from the algebra associated with the L datatype into an appropriate target algebra specified by f [28]. Consider, for example, the prototypical definition of the map^L functional by means of the homomorphic functional hom^L.

$$\begin{array}{l} \mathit{map}^L(A, B \mid \mathit{Type})(f : A \to B) : \ L(A) \to L(B) \\ \qquad ::= \ \ \mathit{hom}^L(\mathit{nil}_B,\ \ \lambda(a,y) : (A \times L(B)).\ f(a) :: y\ ) \end{array}$$

In the rest of this paper we assume the inductive datatypes 0, 1, $\times$, $+$, $\mathbb{B}$, $\mathbb{N}$, and L together with the usual standard operators and relations on these types to be predefined.

N-ary Products and Coproducts. Using higher-order abstraction, type universes, and parametric lists it is possible to internalize n-ary versions of binary type constructors. Consider, for example, the n-ary product $A_1 \times \ldots \times A_n$. It is

[2] Bindings may also employ pattern matching on pairs; for example, a is of type A and l of type $L(A)$ in the binding $(a,l) : (A \times L(A))$.

constructed from the iterator $\otimes(.)$ applied to a list containing the types A_1 through A_n.

$$
\begin{array}{rcl}
\otimes(.) \,:\, L(\mathit{Type}) \to \mathit{Type} & ::= & \mathit{hom}^L(1\,,\,\times) \\
\oplus(.) \,:\, L(\mathit{Type}) \to \mathit{Type} & ::= & \mathit{hom}^L(0\,,\,+)
\end{array}
$$

$\oplus(l)$ represents an n-ary coproduct constructor. Furthermore, pairing may be generalized to tupling, and there is a generalized projection function on tuples $\mathit{proj} : \mathcal{P}$, where

$$
\mathcal{P} \quad ::= \quad \Pi\, l \mid L(\mathit{Type}).\ \otimes(l) \ \to \Pi\, n : \mathbb{N},\ \ p : \ (n < \mathit{len}(l))\,.\ (\mathit{nth}\ \ l\ \ n\ \ p)
$$

The type $(n < m)$ is a proposition which holds if and only if the natural number n is less than m, and the function that selects the nth element of a list is of type $\Pi A \mid \mathit{Type},\ \ l : L(A),\ \ n : \mathbb{N}.\ \ (n < \mathit{len}(l)) \ \to A$. In the definition of proj below abort stands for a term that uses the contradiction $(n < 0)$ to construct a term of the required type.

$$
\begin{array}{l}
\mathit{proj}(l \mid L(\mathit{Type})) :\ \mathcal{P}(l) \quad ::= \\
\quad \mathit{elim}^L_{\mathcal{P}}\ (\lambda\,\_ : 1\,,\ \ n : \mathbb{N},\ \ p : \ (n < 0)\,.\ \mathit{abort}\ \ n\ \ p\ \ (\mathit{nth}\ \ \mathit{nil}_{\mathit{Type}}\ \ n\ \ p)) \\
\qquad\quad (\lambda\, A : \mathit{Type},\ \ l' : L(\mathit{Type}),\ \ y : \mathcal{P}(l'). \\
\qquad\qquad \textsc{let}\ l \quad ::= \quad A :: l'\,, \\
\qquad\qquad\qquad\ C \quad ::= \quad \lambda\, m : \mathbb{N}.\ \Pi\, q : \ (m < \mathit{len}(l))\,.\ (\mathit{nth}\ \ l\ \ m\ \ q) \\
\qquad\qquad \textsc{in}\ \lambda\, a : \otimes(l)\,,\ \ n : \mathbb{N},\ \ p : \ (n < \mathit{len}(l))\,. \\
\qquad\qquad\qquad \mathit{elim}^{\mathbb{N}}_C\ (\lambda\,\_ : \ (0 < \mathit{len}(l))\,.\ \mathit{fst}(a)) \\
\qquad\qquad\qquad\qquad (\lambda\, n : \mathbb{N},\ \ \_ : C(n). \\
\qquad\qquad\qquad\qquad\quad \lambda\, q : \ (\mathit{succ}(n) < \mathit{len}(l))\,.\ (y\ \ \mathit{snd}(a)\ \ n\ \ q)) \\
\qquad\qquad\qquad\qquad n\ \ p)
\end{array}
$$

The main recursion in this definition is along the (hidden) list of types l that are used to describe the generalized product $\otimes(l)$, and the elimination on $\mathit{elim}^{\mathbb{N}}$ runs in parallel to the main recursion. The definition of proj clearly demonstrates that fully synthesized terms are, mainly for the mixture of functional parts and proofs, close to incomprehensible. In these cases, we only state characteristic equations and we omit the proof parts. Using these conventions the definition of the generalized projection function proj simplifies to two equations:

$$
\begin{array}{rcl}
\mathit{proj}\ \ a\ 0 & = & \mathit{fst}(a) \\
\mathit{proj}\ \ a\ \ \mathit{succ}(n) & = & \mathit{proj}\ \ \mathit{snd}(a)\ \ n
\end{array}
$$

The map^L function on lists can be used to generalize $.+.$ and $.\times.$ to also work for the n-ary coproduct $\oplus(.)$ and the n-ary product $\otimes(.)$, respectively. Let $A \mid \mathit{Type}$, $D, R : A \to \mathit{Type}$, $f : \Pi x : A.\ D(a) \to R(a)$, then recursion along the structure of the n-ary type constructors is used to define these generalized mapping functions.

$$\begin{array}{rl} map^{+}(f): & (\Pi l \mid L(A).\ \oplus(map^{L}\ D\ l) \to \oplus(map^{L}\ R\ l)) \\ ::= & elim^{L}\ I_0\ (\lambda(a,l),\ y.\ f(a)+y) \end{array}$$

$$\begin{array}{rl} map^{\times}(f): & (\Pi l \mid L(A).\ \otimes(map^{L}\ D\ l) \to \otimes(map^{L}\ R\ l)) \\ ::= & elim^{L}\ (\lambda\_.\ *)\ (\lambda(a,l),\ y.\ f(a)\times y) \end{array}$$

Although these mappings explicitly exclude tuples with independent types, they are sufficiently general for the purpose of this paper.

3 Semantic Polytypy

In this section we describe a type-theoretic framework for formalizing polytypic programs and proofs. Datatypes are modeled as initial objects in categories of functor-algebras [14], and polytypic constructions—both programs and proofs—are formulated using initiality without reference to the underlying structure of datatypes. We exemplify polytypic program construction in this type-theoretic framework with a polytypic version of the well-known *map* function. Other polytypic developments from the literature (e. g. [2]) can be added easily. Furthermore we generalize some categorical notions to also work for eliminations and lift a reflection and a fusion theorem to this generalized framework. It is, however, not our intention to provide a complete formalization of category theory in type theory; we only define certain categorical notions that are necessary to express the subsequent polytypic developments. For a more elaborated account of category theory within type theory see e. g. [8].

Functors are twofold mappings: they map source objects to target objects and they map morphisms of the source category to morphisms of the target category with the requirement that identity arrows and composition are preserved. Here, we restrict the notion of functors to the category of types (in a fixed, but sufficiently large type universe $Type_i$) with (total) functions as arrows.

Definition 1 (Functor).

$$\begin{array}{l} Functor:\ Type\ ::= \\ \quad \Sigma F_{obj}:\ Type \to Type, \\ \quad\quad F_{arr}:\ \Pi A, B \mid Type.\ (A \to B) \to F_{obj}(A) \to F_{obj}(B). \\ \quad\quad\quad \Pi A \mid Type.\ F_{arr}(I_A) \doteq I_{F_{obj}(A)} \\ \quad\quad \wedge\ \Pi A, B, C \mid Type,\ g: A \to B,\ f: B \to C. \\ \quad\quad\quad\quad F_{arr}(f \circ g) \doteq F_{arr}(f) \circ F_{arr}(g) \end{array}$$

Bifunctors. Generalized functors are functors of type $\overbrace{Type \times \ldots \times Type}^{n \text{ times}} \to Type$; they are used to describe datatypes with n parameter types. In order to keep the technical overhead low, however, we restrict ourselves in this paper to modeling datatypes with one parameter type only by means of bifunctors. Bifunctors are functors of type *Type* $\times$ *Type* $\to$ *Type* or, using the isomorphic curried form, of type *Type* $\to$ *Type* $\to$ *Type*. For a bifunctor, the functor laws in Definition 1 take the following form:

Definition 2 (Bifunctor).

$$
\begin{array}{l}
\mathit{Bifunctor} : \mathit{Type} \quad ::= \\
\quad \Sigma FF_{obj} : \ \mathit{Type} \to \mathit{Type} \to \mathit{Type}, \\
\qquad FF_{arr} : \ \Pi A, B, C, D \mid \mathit{Type}.\ (A \to B) \to (C \to D) \to \\
\qquad\qquad\qquad FF_{obj}\ A\ C \to FF_{obj}\ B\ D. \\
\qquad\quad \Pi A, B \mid \mathit{Type}.\ FF_{arr}\ I_A\ I_B \ \doteq\ I_{FF_{obj}\ A\ B} \\
\qquad \wedge\ \Pi A, B, C, D, E, F \mid \mathit{Type}, \\
\qquad\qquad h : A \to B,\ f : B \to C,\ k : D \to E,\ g : E \to F. \\
\qquad\qquad\quad FF_{arr}(f \circ h)(g \circ k)\ \doteq\ (FF_{arr}\ f\ g) \circ (FF_{arr}\ h\ k)
\end{array}
$$

Many interesting examples of bifunctors are constructed from the unit type and from the product and coproduct type constructors. Seeing parameterized lists as cons-lists with constructors $\mathit{nil}_A : L(A)$ and $\mathit{cons}_A : A \times L(A) \to L(A)$ we get the bifunctor FF^L.

Example 1 (Polymorphic Lists). There exists a proof term p such that:

$$
\begin{array}{l}
FF^L : \ \mathit{Bifunctor} \quad ::= \\
\quad (\lambda\, A, X : \mathit{Type}.\ 1 + (A \times X), \\
\quad\ \lambda\, A, B, X, Y \mid \mathit{Type},\ f : A \to B,\ g : X \to Y.\ I_1 + (f \times g), \\
\quad\ p)
\end{array}
$$

The construction of the proof term p of bifunctoriality closely follows the structure of the definition of $\pi^1(FF^L)$; i.e. the proof uses coproduct induction elim^+ followed by product induction in the $(A \times X)$ case; the resulting base equalities follow trivially from normalization.

Fixing the first argument in a bifunctor yields a functor.

Definition 3. *For all* $(FF_{obj}, FF_{arr}, q) : \mathit{Bifunctor}$ *and* $A : \mathit{Type}$ *there exists a proof term* p *such that*

$$
\begin{array}{l}
\mathit{induced}(FF_{obj}, FF_{arr}, q)(A) : \ \mathit{Functor} \quad ::= \\
\quad (FF_{obj}(A),\ \ \lambda\, X, Y \mid \mathit{Type},\ f : X \to Y.\ FF_{arr}\ I_A\ f,\ p)
\end{array}
$$

Example 2. $F^L : \mathit{Type} \to \mathit{Functor} \quad ::= \quad \mathit{induced}(FF^L)$

Functor Algebras. Using the notion of *Functor* one defines the concept of datatype (see Def. 8) without being forced to introduce a signature, that is, names and typings for the individual sorts (types) and operations involved. The central notion is that of a functor algebra $\mathit{Alg}(F, X)$, where F is a functor. The second type definition below just introduces a name for the signature type $\Sigma^{\mathcal{C}}(F, T)$ of so-called catamorphisms $(\!|\ .\ |\!)$.

Definition 4. *Let F : Functor and X, T : Type; then:*

$$\begin{array}{rcl} Alg(F,X) : \; Type & ::= & F_{obj}(X) \to X \\ \Sigma^{\mathcal{C}}(F,T) : \; Type & ::= & \Pi X \mid Type. \; Alg(F,X) \to (T \to X) \end{array}$$

The initial F-algebra, denoted α, is an initial object in the category of F-algebras. That is, for every F-algebra f there exists a unique object, say $(\!| f |\!)$, such that the following diagram commutes:

$$\begin{tikzcd} F(T) \arrow[r, "\alpha"] \arrow[d, "F(\!| f |\!)"'] & T \arrow[d, "(\!| f |\!)"] \\ F(X) \arrow[r, "f"] & X \end{tikzcd}$$

In the case of lists, the initial $F^L(A)$-algebra α^L is defined by case split (see Section 2) and the corresponding catamorphism is a variant of the homomorphic functional on lists.

Example 3.

$$\begin{array}{rcl} \alpha^L(A \mid Type) : \; Alg(F^L(A), L(A)) & ::= & [nil_A, \; cons_A] \\ (\!| \, . \, |\!)^L(A \mid Type) : \Sigma^{\mathcal{C}}(F^L(A), L(A)) & ::= & \lambda X \mid Type, \; f : Alg(F^L(A), X) \, . \\ & & hom^L(f \circ inl_{1,A\times X}, \; f \circ inr_{1,A\times X}) \end{array}$$

Paramorphisms correspond to structural recursive functions, and can be obtained by computing not only recursive results as is the case for catamorphisms but also the corresponding data structure. The definition of functor algebras and signatures for paramorphisms are a straightforward generalization of Definition 4.

Definition 5. *Let F : Functor and X, T : Type; then:*

$$\begin{array}{rcl} Alg^{\mathcal{P}}(F,T,X) : \; Type & ::= & F_{obj}(T \times X) \to X \\ \Sigma^{\mathcal{P}}(F,T) : \; Type & ::= & \Pi X \mid Type. \; Alg^{\mathcal{P}}(F,T,X) \to (T \to X) \end{array}$$

Any F-algebra f can be lifted to the corresponding notion for paramorphisms using the function $\uparrow(.)$ in Definition 6.

Definition 6. *Let F : Functor, T, X : Type; then:*

$$\uparrow(f : \; Alg(F,X)) : \; Alg^{\mathcal{P}}(F,T,X) \quad ::= \quad f \circ F_{arr}(\pi^2_{T,X})$$

It is well-known that the notions of catamorphisms and paramorphisms are interchangeable, since one can define a paramorphism $\{\!|\, . \,|\!\}$ from a catamorphism $(\!| \, . \, |\!)$ and vice versa.

$$\begin{array}{rcl} (\!| \, . \, |\!) : \; \Sigma^{\mathcal{C}}(F,T) & ::= & \lambda X \mid Type. \{\!|\, . \,|\!\} \circ \uparrow(.) \\ \{\!|\, . \,|\!\} : \; \Sigma^{\mathcal{P}}(F,T) & ::= & \lambda X \mid Type, \; g : \; Alg^{\mathcal{P}}(F,T,X), \; t : T. \\ & & \pi^2((\!| \lambda z : \; F(T \times X). \, (t, \; g(z)) |\!)(t)) \end{array}$$

$$
\begin{tikzcd}
F(T) \arrow[r, "\alpha"] \arrow[d, "F(R_C(f))"'] & T \arrow[d, "R_C(f)"] \\
F(\Sigma x : T.\ C(x)) \arrow[r, "\phi_C(f)"] & \Sigma x : T.\ C(x)
\end{tikzcd}
$$

Fig. 1. Universal Property.

Eliminations, however, are a genuine generalization of both catamorphisms and paramorphisms in that they permit defining functions of dependent type. In this case, the notion of functor algebras generalizes to a type $Alg^{\mathcal{E}}(F, C, \alpha)$ that depends on an F-algebra α, and the signature type $\Sigma^{\mathcal{E}}(F, T, \alpha)$ for eliminations is expressed in terms of this generalized notion of functor algebra.

Definition 7. *Let F : Functor, T : Type, $C : T \to$ Type, $\alpha : Alg(F, T)$; then:*

$$
\begin{aligned}
Alg^{\mathcal{E}}(F, C, \alpha) : \mathit{Type} \quad &::= \quad \Pi z : F_{obj}(\, \Sigma x : T.\ C(x)\,).\ C((\alpha \circ F_{arr}(\pi^1_{T,C}))\ z) \\
\Sigma^{\mathcal{E}}(F, T, \alpha) : \mathit{Type} \quad &::= \quad \Pi C : T \to \mathit{Type}.\ Alg^{\mathcal{E}}(F, C, \alpha) \to \Pi x : T.\ C(x)
\end{aligned}
$$

These definitions simplify to the corresponding notions for paramorphisms (see Def. 5) when instantiating C with a non-dependent type of the form $(\lambda\_ : T.\ X)$. Moreover, the correspondence between the induction hypotheses of the intuitive structural induction rule and the generalized functor algebras $Alg^{\mathcal{E}}(F^L, C, \alpha^L)$ is demonstrated in [19].

The definition of ϕ_C below is the key to using the usual categorical notions like initiality, since it transforms a generalized functor algebra f of type $Alg^{\mathcal{E}}(F, C, \alpha)$ to a functor algebra $Alg(F, \Sigma x : T.\ C(x)\,)$.[3]

Definition 8. *Let F : Functor, T : Type, $\alpha : Alg(F, T)$, elim : $\Sigma^{\mathcal{E}}(F, C, \alpha)$. Furthermore, let $C : T \to$ Type and $f : Alg^{\mathcal{E}}(F, C, \alpha)$; then:*

$$
\phi_C(f) : \quad Alg(F, \Sigma x : T.\ C(x)\,) \quad ::= \quad \langle \alpha \circ F_{arr}(\pi^1_{T,C}),\, f \rangle
$$

Now, we get the initial algebra diagram in Figure 1 [19], where $R_C(f)$ denotes the unique function which makes this diagram commute. It is evident that the first component must be the identity. Thus, $R_C(f)$ is of the form $\langle I,\, elim_C(f) \rangle$ where the term $elim_C(f)$ is used to denote the unique function which makes the diagram commute for the given f of type $Alg^{\mathcal{E}}(F, C, \alpha)$. Altogether, these considerations motivate the following universal property for describing initiality.

[3] Here and in the following the split function $\langle ., . \rangle$ is assumed to also work for function arguments of dependent types; i.e.$\langle f, g \rangle : \ \Sigma x : B.\ C(x) \quad ::= \quad (f(x), g(x))$ where $A, B : \mathit{Type}$, $C : B \to \mathit{Type}$, $f : A \to B$, $g : \Pi x : A.\ C(f(x))$, and $x : A$.

Definition 9.

$$
\begin{array}{l}
\mathit{universal}^{\mathcal{E}}(F, T, \alpha) : \ \mathit{Prop} \quad ::= \\
\quad \Pi\, C : T \to \mathit{Type},\ f : \mathit{Alg}^{\mathcal{E}}(F, C, \alpha)\,. \\
\qquad \exists^1\, E :\ (\Pi\, x : T.\ C(x)). \\
\qquad\quad \text{LET } R_C(f) \quad ::= \quad \langle I,\, E\rangle \text{ IN} \\
\qquad\qquad R_C(f) \circ \alpha \ \doteq\ \phi_C(f) \circ F(R_C(f))
\end{array}
$$

Witnesses of this existential formula are denoted by $\mathit{elim}_C(f)$ *and* $R_C(f)$ *is defined by* $\langle I,\ \mathit{elim}_C(f)\rangle$ *.*

Reflection and Fusion. Eliminations enjoy many nice properties. Here, we concentrate on some illustrative laws like reflection or fusion, but other laws for catamorphisms as described in the literature can be lifted similarly to the case of eliminations.

Lemma 1 (Reflection). *Let* $\mathit{universal}^{\mathcal{E}}(F, T, \alpha)$ *be inhabited; then:*

$$R_{\lambda\_ : T.\ T}(\uparrow(\alpha)) \circ \pi^2_{T,T} \ \doteq\ \mathit{copy}_T$$

where $\mathit{copy}_T \quad ::= \quad \lambda\, x : T.\ (x, x)$

This equality follows directly from uniqueness and some equality reasoning.

$$
\begin{array}{rl}
 & \mathit{copy}_T \circ \alpha \\
\doteq & \langle \alpha \circ F(\pi^1_{T,T}) \circ F(\mathit{copy}_T),\ \alpha \circ F(\pi^2_{T,T}) \circ F(\mathit{copy}_T)\rangle \\
\doteq & \langle \alpha \circ F(\pi^1_{T,T}),\ \alpha \circ F(\pi^2_{T,T})\rangle \circ F(\mathit{copy}_T) \\
\doteq & \phi_C(\uparrow(\alpha)) \circ F(\mathit{copy}_T)
\end{array}
$$

The fusion law for catamorphism is central for many program manipulations [2]. Now, this fusion law is generalized to also work for eliminations.

Lemma 2 (Fusion). *For functor* F *and type* T*, let* $C, D : T \to \mathit{Type}$*,* $f : \mathit{Alg}^{\mathcal{E}}(F, C, \alpha)$*,* $g : \mathit{Alg}^{\mathcal{E}}(F, D, \alpha)$*,* $h : \Sigma x : T.\ C(x) \ \to\ \Sigma x : T.\ D(x)$*, and assume that the following holds:*

$$
\begin{array}{l}
H_1 : \mathit{universal}^{\mathcal{E}}(F, T, \alpha) \\
H_2 : \Pi S : \mathit{Type}, E : S \to \mathit{Type}, u, v : \mathit{Alg}^{\mathcal{E}}(F, E, \alpha)\,. \\
\qquad\qquad u \ \doteq\ v \ \Rightarrow\ \mathit{elim}_E(u) \ \doteq\ \mathit{elim}_E(v)
\end{array}
$$

Then:

$$h \circ \phi_C(f) \ \doteq\ \phi_D(g) \circ F(h) \quad \Rightarrow \quad h \circ R_C(f) \ \doteq\ R_D(g)$$

The proof of this generalized fusion theorem is along the lines of the fusion theorem for catamorphisms [2].

$$
\begin{tikzcd}
F(T) \arrow[r, "F(R_C(f))"] \arrow[d, "\alpha"'] & F(\Sigma x : T.\ C(x)) \arrow[r, "F(h)"] \arrow[d, "\phi_C(f)"] & F(\Sigma x : T.\ D(x)) \arrow[d, "\phi_D(g)"] \\
T \arrow[r, "R_C(f)"'] & \Sigma x : T.\ C(x) \arrow[r, "h"'] & \Sigma x : T.\ D(x)
\end{tikzcd}
$$

This diagram commutes because the left part does (by definition of elimination) and the right part does (by assumption). (Extensional) equality of $h \circ R_C(f)$ and $R_D(g)$ follows from the uniqueness part of the universality property and the functoriality of F. The extra hypothesis H_2 is needed for the fact that the binary relation $\doteq$ (see Section 2) on functions is not a congruence in general.

Catamorphisms and Paramorphisms. A paramorphism is simply a non-dependent version of an elimination scheme and a catamorphism is defined in the usual way from a paramorphism.

Definition 10. *Let $universal^{\mathcal{E}}(F, T, \alpha)$ be inhabited and X : Type; then:*

$$\begin{array}{rcl} \{\!\!\{ . \}\!\!\} : \Sigma^{\mathcal{P}}(F, T) & ::= & \lambda X \mid Type.\ elim_{(\lambda \_ : T.\ X)} \\ (\!| . |\!) : \Sigma^{\mathcal{C}}(F, T) & ::= & \lambda X \mid Type.\ \{\!\!\{ . \}\!\!\} \circ \uparrow (.) \end{array}$$

Lemma 3. *If f : $Alg(F, X)$, g : $Alg^{\mathcal{P}}(F, T, X)$, and $universal^{\mathcal{E}}(F, T, \alpha)$ is inhabited then $\{\!\!\{ . \}\!\!\}$ and $(\!| . |\!)$ are the unique functions satisfying the equations:*

$$\begin{array}{rcl} (\!| f |\!) \circ \alpha & \doteq & f \circ F((\!| f |\!)) \\ \{\!\!\{ g \}\!\!\} \circ \alpha & \doteq & g \circ F(\langle I_T, \{\!\!\{ g \}\!\!\} \rangle) \end{array}$$

Example 4. Let FF : *Bifunctor* then the polytypic *map* function is defined by

$$map(f) : T(A) \to T(B) \quad ::= \quad (\!| \alpha \circ F(f)(I_{T(B)}) |\!)$$

Uniform Specification of Datatypes in ECC. Now, the previous developments are used to specify, in *ECC*, classes of parametric datatypes in a uniform way. The exact extension of this class is unspecified, and we only assume that there is a name, say dt, and a describing bifunctor, say FF_{dt}, for each datatype in this class; see Figure 2. Then it is straightforward to declare three constants T, α, *elim* corresponding to formation, introduction, and elimination rule, respectively (see Figure 2). Thus, we are left with specifying an equality rule. The equality induced by the diagram in Figure 1 is

$$elim_C(f) \circ \alpha \doteq f \circ F(R_C)$$

It gives the following reduction (computation) rule by expressing the equality on the point level and by replacing equality by reducibility ($x : F(T)$):

$$elim_C(f)(\alpha(x)) \rightsquigarrow f(F\ R_C\ x)$$

There is a slight complication, however, since the left-hand side of this reduction rule is of type $C(\alpha(x))$ while the right-hand side is of type $C(\alpha((F(\pi^1_{T,C}) \circ F(R_C(f)))\ x)$. These two types, although provably Leibniz-equal, are not convertible. Consequently, an additional reduction rule, which is justified by the functoriality of F, is needed. This extraneous reduction rule is not mentioned for the extension of *ECC* with datatypes in [19], since the implicit assumption is that a reduction rule is being added for each functor of a syntactically closed class of functors. In this case, the types of the left-hand and right-hand sides are convertible for each instance. Since we are interested in specifying datatypes uniformly, however, we are forced to abstract over the class of all functors, thereby loosing convertibility between these types.

Let $name: \ Type$ and $FF: \ name \to Bifunctor$; then define the induced functor

$$F(dt : name) : Type \to functor \quad ::= \quad induced(FF_{dt})$$

Formation:

$$T \ : \ name \to Type \to Type$$

Introduction:

$$\alpha \ : \ \Pi\, dt \mid name, \ A \mid Type. \ Alg(F_{dt}(A), T_{dt}(A))$$

Elimination:

$$elim \ : \ \Pi\, dt \mid name, \ A \mid Type. \ \Sigma^{\mathcal{E}}(F_{dt}(A), T_{dt}(A), \alpha_{dt,A})$$

Equality: $(C : T \to Type)$

$$\begin{aligned} (F_{dt}(A)(\pi^1_{T,C}) \circ F_{dt}(A)(R_C(f))) &\rightsquigarrow I_{F_{dt}(A)(T_{dt}(A))} \\ elim_C(f)(\alpha(x)) &\rightsquigarrow f(F_{dt}(A)\ R_C\ x) \end{aligned}$$

Fig. 2. Specifying Datatypes in *ECC*

4 Syntactic Polytypy

The essence of polytypic abstraction is that the syntactic structure of a datatype completely determines many developments on this datatype. Hence, we specify a syntactic representation for making the internal structure of datatypes explicit, and generate, in a uniform way, bifunctors from datatype representations. Proofs for establishing the bifunctoriality condition or the unique extension property follow certain patterns. The order of applying product and coproduct inductions in the proof of the existential direction of the unique extension property, for example, is completely determined by the structure of the underlying bifunctor. Hence, one may develop specialized tactics that generate according proofs separately for each datatype under consideration. Here, we go one step further by capturing the general patterns of these proofs and internalizing them in type theory. In this way, polytypic proofs of the applicability conditions of the theory formalized in this section are constructed *once and for all*, and the proof terms for each specific datatypes are obtained by simple instantiation.

These developments are used to instantiate the abstract parameters *name* and *FF* in order to specify a syntactically characterized class of datatypes. Fixing the shape of datatypes permits defining polytypic constructions by recursing (inducting) along the structure of representations. The additional expressiveness is demonstrated by means of defining polytypic recognizers and a polytypic *no confusion* theorem.

In order to keep subsequent developments manageable and to concentrate on the underlying techniques we choose to restrict ourselves to representing the—rather small—class of parametric, polynomial datatypes; it is straightforward, however, to extend these developments to much larger classes of datatypes like the ones described by (strictly) positive functors [5,19]. The only restriction on the choice of datatypes is that the resulting reduction relation as specified in Figure 2 is strongly normalizing. The developments in this section are assumed to be implicitly parameterized over the entities of Figure 2.

A natural representation for polynomial datatypes is given by a list of lists, whereby the j^{th} element in the i^{th} element list determines the type of the j^{th} selector of the i^{th} constructor. The type *Rep* below is used to represent datatypes with n constructors, where n is the length of the representation list, and the type *Sel* restricts the arguments of datatype constructors to the datatype itself (at recursive positions) and to the polymorphic type (at non-recursive positions), respectively. Finally, *rec* and *nonrec* are used as suggestive names for the injection functions of the *Kind* coproduct.

Definition 11 (Representation Types).

$$\begin{array}{rcl} \mathit{Kind} : \mathit{Type} & ::= & \mathit{rec} : 1 + \mathit{nonrec} : 1 \\ \mathit{Sel} : \mathit{Type} & ::= & L(\mathit{Kind}) \\ \mathit{Rep} : \mathit{Type} & ::= & L(\mathit{Sel}) \end{array}$$

Consider the representations dt^L for lists and dt^B binary trees below. The lists *nil* and $(\mathit{nonrec} :: \mathit{rec} :: \mathit{nil})$ in the representation dt^L of the list datatype, for example, describe the signatures of the list constructors *nil* and $(\,.\,::\,.\,)$, respectively.

Example 5.

$$\begin{array}{rcl} dt^L : \mathit{Rep} & ::= & \mathit{nil} :: (\mathit{nonrec} :: \mathit{rec} :: \mathit{nil}) :: \mathit{nil} \\ dt^B : \mathit{Rep} & ::= & \mathit{nil} :: (\mathit{nonrec} :: \mathit{rec} :: \mathit{rec} :: \mathit{nil}) :: \mathit{nil} \end{array}$$

The definitions below introduce, for example, suggestive names for the formation type and constructors corresponding to the representation dt^B in Example 5.

Example 6.

$$B : \mathit{Type} \to \mathit{Type} \quad ::= \quad T(dt^B)$$

$$\mathit{leaf}(A : \mathit{Type}) : \; 1 \to B(A) \quad ::= \quad (\alpha_{dt^B,A} \circ \mathit{inl}_{1,A+B(A)+B(A)+0})$$

$$\mathit{node}(A \mid \mathit{Type}) : \; (A \times B(A) \times B(A) \times 1) \to B(A) \quad ::= \quad (\alpha_{dt^B,A} \circ \mathit{inr}_{1,A+B(A)+B(A)+0} \circ \mathit{inl}_{A+B(A)+B(A),0})$$

Argument types $Arg_{A,X}(s)$ of constructors corresponding to the selector representation s are computed by placing the (parametric) type A at non-recursive positions, type X at recursive positions, and by forming the n-ary product of the resulting list of types.

Definition 12.

$$Arg(A, X : Type)(s : Sel) : \ Type \quad ::= \quad \otimes(map^L \ (rec^+ \ X \ A) \ s)$$

Next, a polytypic bifunctor FF is computed uniformly for the class of representable datatypes. The object part of these functors is easily computed by forming the n-ary sum of the list of argument types (products) of constructors. Likewise the arrow part of FF is computed by recursing over the structure of the representation type Rep. This time, however, the recursion is a bit more involved, since all the types of resulting intermediate functions depend on the form of the part of the representation which has already been processed.

Proposition 1. *For all $dt : Rep$ there exists a proof object p such that*

$$\begin{array}{rcl} FF(dt) : \ Bifunctor & ::= & (FF^{dt}_{obj}, FF^{dt}_{arr}, p) \\ \textit{where } FF^{dt}_{obj} & ::= & (\lambda\, A, X : Type. \ \oplus() \circ map^L(Arg_{A,X})) \ dt \\ FF^{dt}_{arr} & ::= & \lambda\, A, B, X, Y \mid Type, f : A \to B, g : X \to Y. \\ & & \quad (map^+_{Arg_{A,X}, \ Arg_{B,Y}} \circ map^\times_{(rec^+ \ A \ X), \ (rec^+ \ B \ Y)}) \\ & & \qquad (elim^+_{(\lambda\, k:Kind. \ (rec^+ \ A \ X \ k) \to (rec^+ B \ Y \ k))} \ f \ g) \end{array}$$

The inductive construction of the bifunctoriality proof p parallels the structure of the recursive definition of $FF(dt)$. We present one part of this proof—the preservation of identities—in more detail. Let $dt : Rep$, and $A, X : Type$. The goal is to show that

$$FF^{dt}_{arr} \ I_A \ I_X \ \doteq \ I_{FF^{dt}_{obj} \ A \ X}$$

The proof is by induction on the representation type dt. The base case where $dt = nil$ represents an empty datatype and hence the proposition is trivially true. In the induction step one has to prove that

$$\forall\, x : FF^{s::l}_{obj}. \ FF^{s::l}_{arr} \ I_A \ I_X \ x = I_{FF^{s::l}_{obj} \ A \ X} \ x$$

given that the proposition holds for l. The left-hand side of this equation evaluates to

$$\begin{array}{l} (map^\times(elim^+_C \ I_A \ I_X) + FF^l_{arr} \ I_A \ I_X) \ x \\ \textit{where} \quad C \quad ::= \quad \lambda\, k : Kind. \ (rec^+ \ A \ X \ k) \to (rec^+ A \ X \ k) \end{array}$$

We proceed by a coproduct induction on x. The *inr* case can be proved easily using the induction hypothesis. For the *inl* case we have to show that

$$\forall\, y : Arg_{A,X}(s). \ map^\times(elim^+_C \ I_A \ I_X) \ y = I_{Arg_{A,X}(s)} \ y$$

The next step is to induct on the representation s of the argument type of a constructor. The base case corresponds to a zero-place constructor and holds trivially. The induction step requires us to prove that

$$\forall\, y : Arg_{A,X}(k :: l'). \ map^\times_{k::l'}(elim^+_C \ I_A \ I_X) \ y = I_{Arg_{A,X}(k::l')} \ y$$

under the induction hypothesis that the proposition holds for l'. The left-hand side of this equation evaluates to

$$(elim_C^+ \; I_A \; I_X \; k \times map_{l'}^{\times}(elim_C^+ \; I_A \; I_X)) \; y$$

The induction hypothesis reduces the right-hand side function to $I_{Arg_{A,X}(l')}$ and by simple case analysis on k one can prove that

$$elim_C^+ \; I_A \; I_X \; k = I_{rec^+ \; A \; X \; k}$$

Thus, the goal now reads

$$(I_{rec^+ \; A \; X \; k} \times I_{Arg_{A,X}(l')}) \; y = I_{Arg_{A,X}(k::l')} \; y$$

This reduces to the trivial goal

$$(fst(y), snd(y)) = y$$

This polytypic proof has been constructed in Lego using 14 refinement steps.

The second part of the bifunctoriality proof—preservation of composition—runs along the same line. More precisely, the induction proceeds by inducting on the number of coproduct inductions $elim^+$ as determined by the length of the representation type dt followed by an induction on the number of product inductions $elim^\times$ in the induction step; the outer (inner) induction employs one coproduct (product) induction $elim^+$ ($elim^\times$) in its induction step.

n-th Constructor. Example 6 suggests encoding a function for extracting the n-th constructor from α_{dt}. Informally, this function chains n right-injections with a final left-injection:

$$c(n) \equiv \alpha_{dt,A} \circ \underbrace{inr \circ \ldots \circ inr}_{n \text{ times}} \circ inl$$

It is clear how to internalize the ... by recursing on n, but the complete definition of this function is somewhat involved for the dependency of the right injections from the position i and the types of intermediate functions (see also Example 6). Thus, we restrict ourselves to only state the type of this function.

$$\begin{aligned} c : \; & \Pi dt \mid Rep, \; A \mid Type, \; n : \mathbb{N}, \; p : (n < len(dt)) \, . \\ & \quad Arg_{A, T_{dt}(A)} \; (nth \; dt \; n \; p) \to T_{dt}(A) \end{aligned}$$

Polytypic Recognizer. The explicit representation of datatypes permits defining a function for computing recognizer functions for all (representable) datatypes. First, the auxiliary function r traverses a given representation list and returns a pair (f, i) consisting of a recognizer f and the current position i in the representation list.

Definition 13. *Let $dt : Rep$, $A : Type$, $n : \mathbb{N}$, and $p : (n < len(dt))$.*

$$rcg(dt, A, n, p) : Alg(F_{dt}(A), Prop) \quad ::= \quad \pi^1(r^{dt}_{A,n,p})$$

where

$$r(dt, A, n, p) : Alg(F_{dt}(A), Prop) \times \mathbb{N}$$

$$\begin{array}{rcl} r^{nil}_{A,n,p} & = & (\lambda\, x : 0 \, . \; arbitrary_{Prop} \; x, \; zero) \\ r^{s::l}_{A,n,p} & = & \text{LET } (f, i) \; = \; r^{l}_{A,n,p} \text{ IN} \\ & & \quad ([\lambda_{\_} : Arg_{A,Prop}(s). \; n = i, \; f], \; succ(i)) \end{array}$$

Now, it is a simple matter to define the polytypic recognizer by applying the polytypic catamorphism on *rcg*.

Definition 14 (Polytypic Recognizer).

$$R(dt \mid Rep, \; A \mid Type, \; n : \mathbb{N}, \; p : (n < len(dt))) : \; T_{dt}(A) \to Prop \quad ::= \\ (\!| \; rcg^{dt}_{A,n,p} \; |\!)$$

This function satisfies the following characteristic properties for recognizers.

Proposition 2. *Let $dt \mid Rep$, $A \mid Type$, $i, j : \mathbb{N}$; furthermore $p : (i < len(dt))$, $q : (j < len(dt))$, $a : Arg_{A,T_{dt}(A)}(nth \; dt \; i \; p)$, then:*

1. $R_{i,p} \; c_{i,p}(a)$
2. $i \neq j \Rightarrow \neg \, (R_{j,q} \; c_{i,p}(a))$

Polytypic No Confusion. Now, we have collected all the ingredients for stating and proving a polytypic *no confusion* theorem once and for all.

Theorem 1 (Polytypic No Confusion).

$$\begin{array}{l} \Pi\, dt \mid Rep, \; A \mid Type, \; i, j : \mathbb{N}, \; p : (i < len(dt)), \; q : (j < len(dt)), \\ \quad a : \; Arg_{A,T_{dt}(A)} \; (nth \; dt \; i \; p), \; b : \; Arg_{A,T_{dt}(A)} \; (nth \; dt \; j \; q) \\ \qquad i \neq j \Rightarrow c_{i,p}(a) \neq c_{j,q}(b) \end{array}$$

Given the hypothesis $H \; : \; c_{i,p}(a) = c_{j,q}(b)$ one has to construct a proof term of $\bot$. According to Proposition 2 this task can be reduced to finding a proof term for $R_{j,q} \; c_{i,p}(a)$. Furthermore, for hypothesis H, this goal is equivalent with formula $R_{j,q} \; c_{j,q}(a)$, which is trivially inhabited according to Proposition 2. This finishes the proof. Again, each of these steps correspond to a refinement step in the LEGO formalization.

Notice that the proof of the polytypic bifunctoriality property and the polytypic *no confusion* theorem are as straightforward as any instance thereof for any datatype and for any legitimate pair of datatype constructors.

5 Conclusions

The main conclusion of this paper is that the expressiveness of type theory permits internalizing many interesting polytypic constructions that are sometimes thought to be external and not formalizable in the base calculus [22, 24]. In this way, polytypic abstraction in type theory has the potential to add another level of flexibility in the reusability of formal constructions and in the design of libraries for program and proof development systems. We have demonstrated the feasibility of our approach using some small-sized polytypic constructions, but, obviously, much more work needs to be done to build a useful library using this approach. Most importantly, a number of polytypic theorems and polytypic proofs thereof need to be identified.

Although we have used a specific calculus, namely the Extended Calculus of Constructions, for our encodings of *semantically* and *syntactically* polytypic abstraction, similar developments are possible for other type theories such as Martin-Löf type theories [18] with universes or the Inductive Calculus of Constructions.

Semantically polytypic developments are formulated using initiality without reference to the underlying structure of datatypes. We have demonstrated how to generalize some theorems from the literature, like reflection and fusion for catamorphisms, to corresponding theorems for dependent paramorphisms (eliminations). These developments may not only make many program optimizations, like the fusion theorem above, applicable to functions of dependent type but—for the correspondence of dependent paramorphisms with structural induction—it may also be interesting to investigate usage of these generalized polytypic theorems in the proof development process; consider, as a simple example, a polytypic construction of a double induction scheme from structural induction.

Syntactic polytypism is obtained by syntactically characterizing the descriptions of a class of datatypes. Dybjer and Setzer [5] extend the notion of inductively defined sets (datatypes) that are characterized by strictly positive functors [19, 21] and provide an axiomatization of sets defined through induction-recursion. For the purpose of demonstration, however, we have chosen to deal with a special case of datatypes—the class of polynomial datatypes—in this paper, but it is straightforward, albeit somewhat more tedious, to generalize these developments to support larger classes. From a technical point of view, it was essential to be able to recurse along the structure of arbitrary product types in order to encode in type theory many $\ldots$ as used in informal developments. We solved this problem by describing product types $A_1 \times \ldots \times A_n$ by a list $l ::= A_1 :: \ldots :: A_n$ of types and by encoding a function $\otimes(l)$ for computing the corresponding product type. These developments may also be interesting for other applications such as statically typing heterogeneous lists (or S-expressions) within type theory.

The first step towards syntactic polytypism consists in fixing a representation type for the chosen class of datatypes; this representation (or abstract syntax) can be understood as a simple kind of *meta-level* representation, since it makes the internal structure of datatype descriptions amenable for inspection. In the next step, one fixes the denotation of each datatype representation by assigning a corresponding functor to it. These functor terms can be thought of as being on the *object-level* and the type-theoretic function for computing denotations of representation is sometimes called a *computational reflection* function. This internalization of both the representation and the denotation function permits abstracting theorems, proofs, and programs with respect to the class of (syntactically) representable datatypes.

The added expressiveness of syntactic polytypy has been demonstrated by means of several examples. The polytypic (bi)functoriality proof, for example, requires inducting on syntactic representations, and the proof of the *no confusion* theorem relies on the definition of a family of polytypic recognizer functions by recursing on representations. This latter theorem is a particularly good example of polytypic abstraction, since its polytypic proof succinctly captures a 'proof idea' for an infinite number of datatype instances.

References

[1] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre J.C. Fillâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual - Version 6.2.4*. INRIA, Rocquencourt, January 1998.

[2] R. Bird and O. de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.

[3] C. Böhm and A. Berarducci. Automatic Synthesis of Typed λ-Programs on Term Algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[4] Th. Coquand and Chr. Paulin. Inductively Defined Types. In *Proc. COLOG 88*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1990.

[5] P. Dybjer and A. Setzer. A Finite Axiomatization of Inductive-Recursive Definitions. In J.-Y. Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications, TLCA'99*, volume 1581 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[6] M. J. C. Gordon and T. F. Melham (eds.). *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[7] R. Harper and R. Pollack. Type Checking, Universal Polymorphism, and Type Ambiguity in the Calculus of Constructions. In *TAPSOFT'89, volume II*, Lecture Notes in Computer Science, pages 240–256. Springer-Verlag, 1989.

[8] G. Huet and A. Saïbi. Constructive Category Theory. In *Proceedings of the joint CLICS-TYPES Workshop on Categories and Type Theory*, January 1995.

[9] C.B. Jay and J.R.B. Cockett. Shapely Types and Shape Polymorphism. In D. Sannella, editor, *Programming Languages and Systems – ESOP'94*, number 788 in Lecture Notes in Computer Science, pages 302–316. Springer-Verlag, 1994.

[10] J. Jeuring. Polytypic Pattern Matching. In *Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*, pages 238–248, La Jolla, CA, June 1995. ACM Press.

[11] J. Jeuring and P. Jansson. Polytypic Programming. In T. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, Lecture Notes in Computer Science, pages 68–114. Springer-Verlag, 1996.

[12] Z. Luo. An Extended Calculus of Constructions. Technical Report CST-65-90, University of Edinburgh, July 1990.

[13] Z. Luo and R. Pollack. The Lego Proof Development System: A User's Manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

[14] G. Malcolm. Data Structures and Program Transformation. *Science of Computer Programming*, 14:255–279, 1990.

[15] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

[16] L. Meertens. Calculate Polytypically. In H. Kuchen and S.D. Swierstra, editors, *Programming Languages, Implementations, Logics, and Programs (PLILP'96)*, Lecture Notes in Computer Science, pages 1–16. Springer-Verlag, 1996.

[17] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, 1991.

[18] B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Löf's Type Theory*. Number 7 in International Series of Monographs on Computer Science. Oxford Science Publications, 1990.

[19] Ch.E. Ore. The Extended Calculus of Constructions (ECC) with Inductive Types. *Information and Computation*, 99, Nr. 2:231–264, 1992.

[20] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[21] Chr. Paulin-Mohring. Inductive Definitions in the System Coq, Rules and Properties. In J.F. Groote M.Bezem, editor, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 328–345. Springer-Verlag, 1993.

[22] L.C. Paulson. A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definition. Technical report, Computer Laboratory, University of Cambridge, England, 1996.

[23] H. Rueß. Computational Reflection in the Calculus of Constructions and Its Application to Theorem Proving. In J. R. Hindley P. de Groote, editor, *Proceedings of Third International Conference on Typed Lambda Calculi and Applications TLCA'97*, volume 1210 of *Lecture Notes in Computer Science*, pages 319–335. Springer-Verlag, April 1997.

[24] N. Shankar. Steps Towards Mechanizing Program Transformations Using PVS. Preprint submitted to Elsevier Science, 1996.

[25] T. Sheard. Type Parametric Programming. Oregon Graduate Institute of Science and Technology, Portland, OR, USA, 1993.

[26] The LEGO team. The Lego Library. Distributed with Lego System, 1998.

[27] D. Tuijnman. *A Categorical Approach to Functional Programming*. PhD thesis, Universität Ulm, 1996.

[28] F. W. von Henke. An Algebraic Approach to Data Types, Program Verification, and Program Synthesis. In *Mathematical Foundations of Computer Science, Proceedings*. Springer-Verlag Lecture Notes in Computer Science 45, 1976.

[29] F. W. von Henke, S. Pfab, H. Pfeifer, and H. Rueß. Case Studies in Meta-Level Theorem Proving. In Jim Grundy and Malcolm Newey, editors, *Proc. Intl. Conf. on Theorem Proving in Higher Order Logics*, number 1479 in Lecture Notes in Computer Science, pages 461–478. Springer-Verlag, September 1998.