Polytypic Abstraction in Type Theory

H. Pfeifer and H. Rueß

Universität Ulm
Fakultät für Informatik
D-89069 Ulm, Germany
{pfeifer,ruess}@informatik.uni-ulm.de

Abstract. This paper is concerned with formalizations and verifications in type theory that are abstracted with respect to a large class of datatypes; i.e *polytypic* formalizations. The main advantage of these developments are that they 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. Polytypic program and proof construction in a type-theoretic setting is exemplified by the definition of a polytypic *map* function and by mechanized proofs of corresponding properties such as preservation of composition and fusion theorems.

1 Introduction

Many functional programming languages provide a predefined *map* functional on the list datatype. 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.

$$\mathit{map} : \forall A, B.\ (A \to B) \to (\mathit{list}(A) \to \mathit{list}(B))$$

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 datatypes $\mathcal{T}$.

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

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 [Mee92, She93, MFP91, Jeu95, JJ96, Mee96, Tui96]), and concepts from category theory have proven to be especially suitable for expressing polytypic functions and reasoning about them. In this approach, datatypes are modeled as initial objects in categories of functor-algebras [Mal90], and polytypic constructions are formulated using initiality without reference to the underlying structure of datatypes.

This paper examines techniques for expressing polytypic abstraction in type theory. The main advantage of these developments is that they 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. The importance of mechanizing polytypic abstraction is underlined by the fact that large parts of libraries in systems like Coq [CCF+95] or Lego [LP92, JM94] consist of rather trivial developments that have to be repeated and carried out separately for every datatype of interest. This increases the bulk of proving effort and reduces clarity. Formalization of polytypic abstraction in type theory opens the door to proving many useful facts about large classes of datatypes *once and for all*.

[1] Sheard [She93] calls these algorithms *type parametric*, Meertens [Mee92] calls them *generic*, and Jay and Cockett [JC94] 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 datatypes as initial objects in categories of functor-algebras. Polytypic abstraction with respect to the class of polynomial datatypes is exemplified with two well-known fusion theorems. In the rest of the paper we develop techniques to replace the *semantic* condition of initiality in polytypic abstraction with a syntactic one. The main idea is to use representations that make the internal structure of datatypes explicit, and to compute type-theoretic specifications for all represented datatypes in a uniform way; developments abstracted with respect to datatype representations are said to be syntactically polytypic. Section 4 describes a simple representation of datatypes and Section 5 includes the details of syntactic polytypic abstraction. Section 6 concludes with some remarks.

All constructions in this paper have been developed and checked with the LEGO [LP92] system. For the sake of readability we present typeset versions of the original LEGO terms and we take the freedom to use some syntactic conventions—such as infix notation—not available in LEGO.

2 Preliminaries

Our starting point is the *Extended Calculus of Constructions* [Luo90] (*ECC*) enriched with some inductive datatypes as implemented in the LEGO system. The main purpose of this section is to sketch basic concepts of this type theory, to fix the notation, and to discuss the treatment of datatypes in constructive type theory.

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 $\to$ associates to the right. Types of the form $\Sigma x : A.\ B(x)$ comprise dependent pairs. Moreover, 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 [Luo90]. 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 [HP89] and omit subscripts i of type universes Type_i.

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. Definitions like $c(x_1 : A_1, \ldots x_n : A_n) :\ B \ ::=\ M$ are used to introduce a name c for the term M (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 and function composition.

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

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

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 [LP92]).

Using the principle of *propositions-as-types*, the dependent product type $\Pi x : A.\ B(x)$ is interpreted as logical universal-quantification and $\lambda\, x : A.\ M$ is interpreted as a proof term for the formula $\Pi x : A.\ B$. It is possible to encode in *ECC* all the usual logical connectives and quantifiers ($\top$, $\bot$, $\forall$, $\exists$, $\wedge$, $\neg$, $\Rightarrow$, $\ldots$) 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 (and by reduction rules for datatypes, see below). Constructions in this paper employ, besides Leibniz equality, a (restricted) form of extensional equality $. \doteq .$ 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 [BB85]. 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 [NPS90, CP90, PM93, JM94]. 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.

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

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.

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

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

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

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

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

Moreover, the (overloaded) $. \times .$ function plays a central role in categorical specifications of datatypes.

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

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".

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

Similar to the case of products, the symbol $+$ is overloaded to also denote the bifunctor (see Def. 2 below) on coproducts.

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

Unlike products or coproducts, the datatype of parametric lists with constructors *nil* and *cons* is an example of a genuinely recursive datatype.

$$\begin{aligned} \mathit{list} \;&:\; \mathit{Type} \to \mathit{Type} \\ \mathit{nil} \;&:\; \mathit{\Pi} A : \mathit{Type}.\ \mathit{list}(A) \\ \mathit{cons} \;&:\; \mathit{\Pi} A \mid \mathit{Type}.\ A \times \mathit{list}(A) \to \mathit{list}(A) \end{aligned}$$

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

$$\begin{aligned} \mathit{elim}^{\mathit{list}} \; : \; & \Pi A \mid \mathit{Type},\ C : \mathit{list}(A) \to \mathit{Type}, \\ & (C(\mathit{nil}_A) \times (\Pi(a,l) : (A \times \mathit{list}(A)).\ C(l) \to C(\mathit{cons}(a,l))) \\ & \quad \to \Pi l : \mathit{list}(A).\ C(l) \end{aligned}$$

and of the rewrites corresponding to equality rules:

$$\begin{aligned} \mathit{elim}^{\mathit{list}}_C\ f\ \mathit{nil}_A &\rightsquigarrow \mathit{fst}(f) \\ \mathit{elim}^{\mathit{list}}_C\ f\ \mathit{cons}(a,l) &\rightsquigarrow \mathit{snd}(f)\ (a,l)\ (\mathit{elim}^{\mathit{list}}_C\ f\ l) \end{aligned}$$

The non-dependent variants $\mathit{rec}^{\mathit{list}}$ and $\mathit{hom}^{\mathit{list}}$ of list elimination are used to encode structural recursive functions on lists.

$$\begin{aligned} & \mathit{rec}^{\mathit{list}}(A, C \mid \mathit{Type},\ f : C \times ((A \times \mathit{list}(A)) \to C \to C)) \; : \; \mathit{list}(A) \to C \quad ::= \\ & \quad \mathit{elim}^{\mathit{list}}(\lambda\_ : \mathit{list}(A).\ C)(f) \end{aligned}$$

$$\begin{aligned} & \mathit{hom}^{\mathit{list}}(A, C \mid \mathit{Type},\ f : C \times ((A \times C) \to C)) :\ \mathit{list}(A) \to C \quad ::= \\ & \quad \mathit{rec}^{\mathit{list}}(\mathit{fst}(f),\ \lambda(a,\_) : A \times \mathit{list}(A),\ y :\ C.\ \mathit{snd}(f)(a,y)) \end{aligned}$$

The name $\mathit{hom}^{\mathit{list}}$ stems from the fact that $\mathit{hom}^{\mathit{list}}(f)$ can be characterized as a (unique) homomorphism from the algebra associated with the *list* datatype into an appropriate target algebra specified by f [vH76]. Consider, for example, the prototypical definition of the *map* functional by means of the homomorphic functional $\mathit{hom}^{\mathit{list}}$.

$$\begin{aligned} & \mathit{map}^{\mathit{list}}(A, B \mid \mathit{Type},\ f : A \to B) :\ \mathit{list}(A) \to \mathit{list}(B) \quad ::= \\ & \quad \mathit{hom}^{\mathit{list}}(\mathit{nil}_B,\ \lambda(a,y) : (A \times \mathit{list}(B)).\ \mathit{cons}(f(a), y)) \end{aligned}$$

In the rest of this paper we assume the inductive datatypes $\mathbf{0}$, $\mathbf{1}$, $\times$, $+$, and *list* 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 constructed from the iterator $\otimes(.)$ applied to a list containing the types A_1 through A_n.

$$\begin{aligned} \otimes(.) \; : \; \mathit{list}(\mathit{Type}) \to \mathit{Type} \quad &::= \quad \mathit{hom}^{\mathit{list}}(\mathbf{1},\ \times) \\ \oplus(.) \; : \; \mathit{list}(\mathit{Type}) \to \mathit{Type} \quad &::= \quad \mathit{hom}^{\mathit{list}}(\mathbf{0},\ +) \end{aligned}$$

$\oplus(l)$ represents an n-ary coproduct constructor. Furthermore, pairing may be generalized to tupling, there is a generalized projection function on tuples, and, most interestlingly, the bifunctors $\times$ and $+$ can be generalized to the n-ary functors $\otimes(.)$ and $\oplus(.)$ respectively.

3 Semantic Polytypy

In this section we describe a type-theoretic framework for formalizing polytypic programs and program transformations. Datatypes are modeled as initial objects in categories of functor-algebras [Mal90], 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 the polytypic *map* function, corresponding properties such as preservation of composition, and polytypic fusion theorems; other polytypic developments from the literature can be added easily.

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

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.

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

Moreover, $\mathit{functorial}(F_{obj})(F_{arr})$ denotes the conjunction of the two preservation properties. □

Bifunctors are functors of type $\mathit{Type} \times \mathit{Type} \to \mathit{Type}$ or, using the equivalent curried form, of type $\mathit{Type} \to \mathit{Type} \to \mathit{Type}$; they are used to describe polymorphic datatypes. For a bifunctor, the functor laws in Def. 1 take the following form.

Definition 2 Bifunctor.

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

The shorthand $\mathit{bifunctorial}(F_{obj})(F_{arr})$ is used to denote the preservation properties of bifunctors. □

Many interesting examples of bifunctors are constructed from type constructors like $\mathbf{1}$, product, and coproduct. Seeing parameterized lists as cons-lists with constructors $\mathit{nil}_A : \mathit{list}(A)$ and $\mathit{cons}_A : A \times \mathit{list}(A) \to \mathit{list}(A)$ we get the bifunctor FF^{list}.

Example 3 Polymorphic Lists.

$$FF^{list}_{obj}(A, X : \mathit{Type}) : \ \mathit{Type} \ ::= \ \mathbf{1} + (A \times X)$$

$$FF^{list}_{arr}(A, B, C, D \mid \mathit{Type})(f : A \to B)(g : C \to D) : \ (FF^{list}_{obj}\ A\ C) \to (FF^{list}_{obj}\ B\ D) \ ::= \ I_{\mathbf{1}} + (f \times g)$$

The proof that the proposition ($\mathit{bifunctorial}\ FF^{list}_{obj}\ FF^{list}_{arr}$) is inhabited closely follows the structure of the definition of FF^{list}_{obj}; 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. The dot notation is used to project from elements of Σ-types and obj, arr name the first and second projections, respectively, on (bi)functors.

Proposition 4. $\Pi FF : \mathit{Bifunctor},\ A : \mathit{Type}.\ \mathit{functorial}\ FF.\mathit{obj}(A)\ FF.\mathit{arr}(I_A)$

Fusion and Reflection in Initial Algebras. Using the notion of *functor* one defines the concept of datatype (algebra) as in Def. 5 without being forced to introduce a signature, that is, names and typings for the individual sorts (types) and operations involved. An F-algebra for an arbitrary functor F is a function f of type $F\ A \to A$. The existence of an initial F-algebra, say α, means that for any other F-algebra f there is a unique homomorphism, say $\mathit{cata}(f)$, from α to f; i.e. $\mathit{cata}(f)$ is the only F-algebra such that $\mathit{cata}(f) \circ \alpha \doteq f \circ F_{arr}(\mathit{cata}(f))$. Definition 5 makes these statements precise.

Definition 5 Initial Datatypes. Given the type declarations $T : \mathit{Type}$, $F : \mathit{Functor}$, and

$$\begin{aligned}\alpha &: F.\mathit{obj}(T) \rightarrow T\\ \mathit{cata} &: \Pi A \mid \mathit{Type}.\ (F.\mathit{obj}(A) \rightarrow A) \rightarrow (T \rightarrow A)\end{aligned}$$

we say that α is the initial F-algebra (with respect to cata) if the proposition

$$\begin{aligned}&\mathit{unique\_extension}(T)(F)(\alpha)(\mathit{cata}) : \ \mathit{Prop} \ ::=\\ &\quad \Pi A \mid \mathit{Type},\ h : T \rightarrow A,\ f : F.\mathit{obj}(A) \rightarrow A.\ h \circ \alpha \doteq f \circ F.\mathit{arr}(h) \ \Leftrightarrow\ h \doteq \mathit{cata}(f)\end{aligned}$$

is inhabited; then, $\mathit{cata}(f)$ is called a *catamorphism*.

Catamorphisms enjoy many nice properties like reflection or fusion laws, and α is an isomorphism with $\mathit{cata}(F.\mathit{arr}(\alpha))$ as functional inverse.

Lemma 6. *Given the type declarations for T, F, α, and cata in Def. 5 together with the hypotheses*

$$\begin{aligned}H_1 &: \mathit{unique\_extension}(T)(F)(\alpha)(\mathit{cata})\\ H_2 &: \Pi A \mid \mathit{Type},\ f, g : F.\mathit{obj}(A) \rightarrow A.\ f \doteq g \ \Rightarrow\ \mathit{cata}(f) \doteq \mathit{cata}(g)\end{aligned}$$

the following formulas are inhabited.

$$\mathit{cata}(\alpha) \doteq I_T \qquad \text{(Reflection)}$$

$$\begin{aligned}&\Pi A, B \mid \mathit{Type}, f : F.\mathit{obj}(A) \rightarrow A, g : F.\mathit{obj}(B) \rightarrow B, h : A \rightarrow B.\\ &\quad h \circ f \doteq g \circ F.\mathit{arr}(h) \ \Rightarrow\ h \circ \mathit{cata}(f) \doteq \mathit{cata}(g)\end{aligned} \qquad \text{(Fusion)}$$

$$\alpha \circ \mathit{cata}(F.\mathit{arr}(\alpha)) \doteq I_T\ ,\ \ \mathit{cata}(F.\mathit{arr}(\alpha)) \circ \alpha \doteq I_{F.\mathit{obj}(T)} \qquad \text{(Lambek's lemma)}$$

The mechanized proofs of these properties are along the line of published equational proofs [BdM97]. For the fact that the equality $\doteq$ (see Section 2) on functions is not a congruence relation on terms, however, proofs are on the point level and the condition H_2 is required to replace the functional argument in $\mathit{cata}(f)$ with $\mathit{cata}(g)$ when $f \doteq g$. In the case of lists, the initial $\mathit{FF}^{\mathit{list}}(A)$-algebra is defined by case split (see Section 2) and the corresponding catamorphism is a variant of the homomorphic functional on lists.

Example 7.

$$\begin{aligned}\alpha^{\mathit{list}}(A \mid \mathit{Type}) : \ (\mathit{FF}^{\mathit{list}}\ A\ \mathit{list}(A)) \rightarrow \mathit{list}(A) \ &::= \ [\mathit{nil}_A,\ \mathit{cons}_A]\\ \mathit{cata}^{\mathit{list}}(A, B \mid \mathit{Type})(f : \ (\mathit{FF}^{\mathit{list}}\ A\ B) \rightarrow B) : \ \mathit{list}(A) \rightarrow B \ &::= \ \mathit{hom}^{\mathit{list}}(f \circ \mathit{inl}_{\mathbf{1}, A\times B},\ f \circ \mathit{inr}_{\mathbf{1}, A\times B})\end{aligned}$$

According to Proposition 4 fixing the first argument of a bifunctor yields a functor. This fact is used to define the induced type functor from a bifunctor FF with the collection of initial algebras α_A.

Definition 8 Induced Type Functor. Given the type declarations $T : \ \mathit{Type} \rightarrow \mathit{Type}$, $\mathit{FF} : \ \mathit{Bifunctor}$, and

$$\begin{aligned}\alpha &: \Pi A \mid \mathit{Type}.\ (\mathit{FF}.\mathit{obj}\ A)\ T(A) \rightarrow T(A)\\ \mathit{cata} &: \Pi A, B \mid \mathit{Type}.\ (((\mathit{FF}.\mathit{obj}\ A)\ B) \rightarrow B) \rightarrow (T(A) \rightarrow B)\end{aligned}$$

the *induced type functor* T_{arr} is defined as follows:

$$T_{\mathit{arr}}(A, B \mid \mathit{Type})(f : A \rightarrow B) : \ T(A) \rightarrow T(B) \ \ ::= \ \ \mathit{cata}(\mathit{alpha}_B \circ (\mathit{FF}.\mathit{arr}\ f\ I_{T(B)}))$$

Now, Theorem 9 states the well-known fact that indeed T, T_{arr} form a functor. The proof of this fact uses so-called *type functor fusion*, which can be regarded as an optimizing transformation that permits merging two iterations of the form $\mathit{cata}(h) \circ T_{\mathit{arr}}(g)$ into a single iteration. The essential step in the proof of type functor fusion is the application of fusion in Lemma 6.

Theorem 9. *Given the type declarations for T, FF, α, and cata in Def. 8 and the assumptions*

$$H_1 : \Pi A \mid Type.\ unique\_extension\ (T\ A)\ (FF.obj\ A)\ (FF.arr\ I_A)\ \alpha_A\ cata_A$$
$$H_2 : \Pi A, B \mid Type,\ f, g : ((FF.obj\ A)\ B \to B).\ f \doteq g \Rightarrow cata(f) \doteq cata(g)$$

the following propositions are inhabited:

$$\Pi A, B, C \mid Type,\ h : (FF.obj\ B\ C) \to C,\ g : A \to B. \qquad \textit{(Type Functor Fusion)}$$
$$cata(h) \circ T_{arr}(g) \doteq cata(h \circ (FF.arr\ g\ I_C))$$

$$functorial\ T\ T_{arr} \qquad \textit{(Type Functor)}$$

The formal proofs of these facts are straightforward transliterations of the developments in Chapter 2 of [BdM97]. Coming back to our running example, it is easy to see that the prerequisites of Theorem 9 are fulfilled for parametric lists.

$$\Pi A : Type.\ unique\_extension\ list(A)\ FF^{list}_{obj}(A)\ FF^{list}_{arr}(I_A)\ \alpha^{list}_A\ cata^{list}_A$$

$$\Pi A, B : Type,\ f, g : A \to B.\ f \doteq g \Rightarrow cata^{list}(f) \doteq cata^{list}(g)$$

The *existence* part of the unique extension property is proved by induction on the structure of FF^{list} (as in the bifunctoriality proof for FF^{list} above), while both the *uniqueness* part and the equality property for $cata(f)$ require list induction $elim^{list}$. In this way, one obtains, through simple instantiation, the usual *map* functional on parametric lists together with proof (terms) of the facts that this *map* function preserves both identities and composition.

More importantly, 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.

4 Representation of Datatypes

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. In order to keep subsequent developments manageable and to concentrate on the underlying techniques we chose to restrict ourselves to representations of the–rather small—class of parametric, polynomial datatypes; similar developments, however, should be possible to deal with larger classes.

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). Finally, *rec* and *nonrec* are used as suggestive names for the injection functions of the *Kind* coproduct.

Definition 10 (Representation Types).

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

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

Example 11.

$$\begin{array}{rcl} DT^{list} : Rep & ::= & nil_{Kind} :: (nonrec :: rec) \\ DT^{btree} : Rep & ::= & nil_{Kind} :: (nonrec :: rec :: rec) \end{array}$$

The term $Constr(A, X, l)$ denotes a schematic type for the constructor represented by the list l. The corresponding argument type $Arg(A, X, l)$ is computed from the representation l by placing type A at nonrecursive positions, type X at recursive positions, and by forming the n-ary product of the resulting list of types.

Definition 12.

$$\begin{array}{rcl} Arg(A, X : Type, l : Sel) : Type & ::= & \otimes(map\ (rec^{+}\ X\ A)\ l) \\ Constr(A, X : Type, l : Sel) : Type & ::= & (Arg\ A\ X\ l) \to X \end{array}$$

Next, a polytypic bifunctor FF^{poly} 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.

Definition 13.

$$FF^{poly}_{obj}(dt : Rep,\ A, X : Type) : Type \ ::= \ \oplus(map\ (Arg\ A\ X)\ dt)$$

Likewise, the arrow part FF^{poly}_{arr} is computed by recursing over the structure of the representation type *Rep* of datatypes; i.e. by recursing on the outer list, the inner lists, and by case analysis on the elements of type *Kind*. This time, however, the recursion is a bit more involved, since the resulting function type depends on the representation itself.

Definition 14. Let $dt : Rep,\ A, B, X, Y \mid Type,\ g : A \to B$, and $f : X \to Y$; then:

$$\begin{array}{l} FF^{poly}_{arr}(dt,\ A, B, X, Y, g, f) : (FF^{poly}_{obj}\ dt\ A\ X) \to (FF^{poly}_{obj}\ dt\ B\ Y) \ ::= \\ \quad elim^{list}\ (\lambda\, l : Rep.\ (FF^{poly}_{obj}\ l\ A\ X) \to (FF^{poly}_{obj}\ l\ B\ Y)) \\ \qquad (I_{\mathbf{0}},\ \ \lambda\, a : Sel, l : Rep, y : (FF^{poly}_{obj}\ l\ A\ X) \to (FF^{poly}_{obj}\ l\ B\ Y). \\ \qquad\qquad\qquad (prod_{arr}\ g\ f\ l\ a) + y) \\ \qquad dt \end{array}$$

$$\begin{array}{l} prod_{arr}(A, B, X, Y, g, f) : Sel \to (Arg\ A\ X\ dt) \to (Arg\ B\ Y\ dt) \ ::= \\ \quad elim^{list}\ (\lambda\, l : Sel.\ (Arg\ A\ X\ l) \to (Arg\ B\ Y\ l)) \\ \qquad (I_{\mathbf{1}},\ \lambda\, a : Arg,\ l : Sel,\ y : (Arg\ A\ X\ l) \to (Arg\ B\ Y\ l). \\ \qquad\qquad\qquad elim^{+}\ (\lambda\, k : Kind.\ (Arg\ A\ X\ cons(k, l)) \to (Arg\ B\ Y\ cons(k, l))) \\ \qquad\qquad\qquad\quad (f \times y)\ (g \times y)\ a) \square \end{array}$$

Recall that elimination on lists is of the form $elim^{list}(C)(f_{nil}, f_{cons})$, where C determines the (dependent) target type and f_{nil}, f_{cons} specify the computations in the *nil* and in the *cons* case, respectively. Application of FF^{poly}_{arr} yields the identity function on the $\mathbf{0}$ type in case of an empty datatype representation; otherwise it sums the function $(prod_{arr}\ g\ f\ a)$ with the function y as accumulated by recursive calls. This product function is defined in a similar fashion by inducting on the list of selectors followed by case analysis: if the current argument a describes a recursive (nonrecursive) position then one multiplies f (g) with the recursively computed function y.

It is not hard to verify that FF^{poly}_{obj}, FF^{poly}_{arr} preserve identities and composition for every possible datatype representation dt by generalizing the proof for establishing bifunctoriality of FF^{list} in Section 3.

Proposition 15. $\Pi\, dt : Rep.\ bifunctorial\ FF^{poly}_{obj}(dt)\ FF^{poly}_{arr}(dt)$

The inductive proof of this fact parallels the structure of the recursive definition of $FF^{poly}_{arr}(dt)$. More precisely, this 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. Thus, tupling the terms $FF^{poly}_{obj}(dt)$, $FF^{poly}_{arr}(dt)$, and the proof term constructed above yields an element, say $FF^{poly}(dt)$ of type *Bifunctor* (see Definition 2).

5 Syntactic Polytypy

For the class of representable datatypes dt as introduced in Section 4, we now extend the underlying type theory with four (families of) constants T_{dt}, $intro_{dt}$, $elim_{dt}$, eqs_{dt} corresponding to the formation, introduction, elimination, and equality rules of parametric, polynomial datatypes. These constant declarations are completely internalized in that the types of the declared constants are computed, by means of type-theoretic functions, in a uniform way from datatype representations dt.

Definition 16.

$$\begin{aligned}
T\ &:\ rep \to Type \to Type\\
intro\ &:\ \Pi\, dt : Rep,\ A : Type.\ (FF^{poly}_{obj}\ dt\ A\ T_{dt}(A)) \to T_{dt}(A)\\
elim\ &:\ \Pi\, dt : Rep,\ A : Type,\ C : T_{dt}(A) \to Type.\ (IndSteps\ dt\ A\ C) \to \Pi x : T_{dt}(A).\ C(x)\\
eqs\ &:\ \Pi\, dt : Rep,\ A : Type,\ C : T_{dt}(A) \to Type,\ f : (IndSteps\ dt\ A\ C).\ Equalities\ dt\ A\ C\ f
\end{aligned}$$

The family of constants T_{dt} can be regarded as names for parametric, polynomial datatypes, while the polytypic bifunctor FF^{poly}_{obj} (Def. 13) determines the type of the constructors $intro_{dt}$ of the datatype T_{dt}. The definitions below introduce, for example, suggestive names for the formation type and constructors corresponding to the representation DT^{btree} as defined in Example 11.

$$btree :\ Type \to Type\ \ ::=\ \ T(DT^{btree})$$

$$leaf(A : Type) :\ \mathbf{1} \to btree(A)\ \ ::=\ \ (intro\ DT^{btree}\ A) \circ inl_{\mathbf{1},A\times btree(A)\times btree(A)+\mathbf{0}}$$

$$\begin{aligned}
&node(A \mid Type) :\ (A \times btree(A) \times btree(A) \times \mathbf{1}) \to btree(A)\ \ ::=\\
&\quad (intro\ DT^{btree}\ A) \circ inr_{\mathbf{1},A\times btree(A)\times btree(A)+\mathbf{0}} \circ inl_{A\times btree(A)\times btree(A),\,\mathbf{0}}
\end{aligned}$$

Next, constant *elim* in Definition 16 specifies the polytypic elimination rule. The conjunction of induction steps ($IndSteps\ dt\ A\ C$) in the type of *elim* are computed by recursing over the datatype representation dt as in Definition 14. The type of $elim(DT^{btree})$, for example, is convertible with

$$\begin{aligned}
&\Pi A : Type,\ C : T_{dt}(A) \to Type.\\
&\quad (C(leaf(A)) \times (\Pi(a,l,r,\_) :\ (A \times btree(A) \times btree(A) \times \mathbf{1}).\\
&\qquad\qquad ((C(l) \times C(r) \times \mathbf{1}) \to C(node(a,l,r)))) \times \mathbf{1})\\
&\qquad \to \Pi x : T_{dt}(A).\ C(x)
\end{aligned}$$

Details of the recursive definitions for computing types corresponding to induction steps in elimination rules and of the conjunctions of equality rules can be found in Appendix A. Notice also that these definitions only rely on the presence of the datatypes $\mathbf{0}$, $\mathbf{1}$, product, coproduct, and polymorphic lists.

Hence, one may conclude that the declared constants T, *intro*, *elim*, and *eqs* characterize the class of parametric, polynomial datatypes. This extension of the base calculus with new constants can be viewed as an internalization of what is usually achieved by introducing new typing rules to the underlying calculus. In the rest of this section it is demonstrated how internalized datatype representations can be used to specify *syntactically polytypic* constructions; i.e. constructions abstracted with respect to a class of representable datatypes.

Definition 17.

$$rec(dt : Rep, A : Type, Y \mid Type) \;\; ::= \;\; elim_{dt}\ A\ (\lambda\_ : T_{dt}(A).\ Y)$$

$$\begin{aligned}&cata(dt \mid Rep, A, X \mid Type)(f : FF^{poly}_{obj}(dt)\ A\ X) : \ T_{dt}(A) \to X \;\; ::= \\ &\quad rec\ dt\ A\ (insert\_recs\ dt\ A\ f\ (intro_{dt}(A)))\end{aligned}$$

The polytypic recursor *rec* is defined in the usual way as the non-dependent variant of elimination and *cata* is defined from *rec* by transforming collections of functions of type $FF^{poly}_{obj}(dt)\ A\ X$ to elements of type $IndSteps\ dt\ A\ C$; this transformation *insert_recs* is easily defined by means of recursing on the structure of representations.

Now we have collected all the ingredients to use semantically polytypic developments like the ones in Section 3 to define syntactically polytypic functions, since initial algebras and catamorphisms correspond to polytypic constructors $intro_{dt}$ and the polytypic catamorphism $cata_{dt}$, respectively. Consider, for example, the instantiation of the induced type functor T_{arr} (Definition 8) for defining the syntactically polytypic *map* function.

Example 18 Polytypic Map.

$$\begin{aligned}&map(A, B \mid Type,\ f : A \to B) : \ \Pi\, dt \mid Rep.\ T_{dt}(A) \to T_{dt}(B) \;\; ::= \\ &\quad (T_{arr}\ T_{dt}\ FF^{poly}_{dt}\ intro_{dt}\ cata_{dt})\ f\end{aligned}$$

For the hidden binding $dt \mid Rep$ in the definition of *map*, LEGO's inference mechanism may be used to automatically infer the appropriate datatype representation *dt* when applying the *map f* function to elements of type $T_{dt}(A)$. For example, LEGO is capable of inferring the hidden argument DT^{btree} for representing binary trees from the application $(map\ f\ leaf(a, l, r))$, since $leaf(a, l, r)$ is of type $T(DT^{btree})$. Hence, the effect of synthesizing hidden datatype representations in LEGO is similar to the one obtained by extending type systems with a polytypic construct as described in [Jan97]. Moreover, by applying Theorem 9, the polytypic function *map* can readily be shown to be functorial and to satisfy equations like induced type functor fusion, since $intro_{dt}$ are initial algebras with respect to the catamorphisms $cata_{dt}$ (and $cata_{dt}$ satisfies the extraneous extensionality property required in Theorem 9).

6 Conclusions

We have shown how to formalize two conceptually differing notions of polytypism, namely semantic and syntactic polytypism, in a type-theoretic setting. Semantically polytypic developments like the type functor fusion theorem are formulated using initiality without reference to the underlying structure of datatypes. On the other hand, syntactic polytypism is obtained by fixing a certain class of datatypes such a the class defined by polynomial functors. We internalize the description of polynomial datatypes in type theory by defining a number of type-producing functions in type theory for computing, in a uniform fashion, datatype specifications corresponding to introduction, formation, elimination, and equality rules. This internalization permits abstracting theorems, proofs, and programs with respect to the class of representable datatypes. Furthermore, since every polynomial datatype is expressible as the initial object in a functor algebra, it is possible to instantiate semantically polytypic developments for this syntactically specified class of datatypes *once and for all.* This fact has been used in this paper to instantiate, in a formal setting, program transformations like fusion theorems for all polynomial datatypes. In addition to the examples described in this paper, we have also experimented with expressing and proving standard properties on datatypes like a (syntactically) polytypic freeness theorem.

The constructions in this paper have been developed and checked with the LEGO system [LP92, Pol94]; This system implements the Extended Calculus of Constructions, but similar encodings should be possible for other type theories such as Martin-Löf type theories [NPS90] with universes or the inductive calculus of constructions. In our experience, it has been rather straightforward to encode semantic polytypy in type theory and to transliterate equational proofs of properties like fusion theorems from the unique extension property. For the lack of extensional equality in LEGO, however, equality proofs have to be performed at the point level. Moreover, it proved to be surprisingly difficult to internalize polytypic abstraction with respect to the—rather small—class of polynomial datatypes; this complication is mainly due to the restriction to structural recursion and the clumsiness of defining recursive functions with dependent types.

A lot of work remains to be done to internalize a larger class of datatypes including mutually recursive datatypes, infinitely-branching datatypes, and codatatypes; good starting points seem to be the extensions of type-theoretic calculi with inductive types as described by Ore [Ore92] or Paulin-Mohring [PM93].

Shankar's [Sha96] verification of a fusion theorem due to [Bir95] shows that current theorem-proving capabilities can effectively be used to formalize and verify advanced program transformations. His analysis, however, is not polytypic and restricted to one specific datatype. In Paulson's [Pau96] extension of the Isabelle system, (co)inductive datatypes are specified as fixedpoints of monotonic predicate transformers, and tactics construct, from such a datatype definition, proofs of theorems corresponding to introduction and (co)induction rules. Although structural recursors are omitted in Paulson's package it is still more practical and powerful than ours in that it supports basically all datatypes of interest. On the other hand, Paulson's package is written in a meta-language and functions in this package must be executed separately for each datatype under consideration, while our formalizations are internalized in type theory and permit expressing (syntactically) polytypic theorems and interactive proof construction for these theorems.

The main conclusion of this paper is that the expressiveness of type theory can be used to internalize many interesting polytypic program development steps and theorems that are usually thought to be meta constructions and meta theorems. In this way, polytypic abstraction in type theory has the potential to add another level of flexibility in the reusability of formal proofs and in the design of libraries for program and proof development systems.

References

[BB85] C. Böhm and A. Berarducci. Automatic synthesis of type λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[BdM97] R. Bird and O. de Moor. *Algebra of Programming.* International Series in Computer Science. Prentice Hall, 1997.

[Bir95] R.S. Bird. Functional Algorithm Design. In B. Möller, editor, *Mathematics of Program Construction '95*, Lecture Notes in Computer Science, pages 2–17. Springer, 1995.

[CCF$^+$95] C. Cornes, J. Courant, J.C. Fillâtre, G. Huet, P. Manoury, C. Muñoz, Ch. Murthy, C. Parent, Chr. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual - Version 5.10.* INRIA, Rocquencourt, July 1995.

[CP90] 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.

[HP89] 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.

[Jan97] P. Jansson. *Functional Polytypic Programming: Use and Implementation.* PhD thesis, Department of Computer Science, Chalmers University of Technology, 1997.

[JC94] 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.

[Jeu95] 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.

[JJ96] 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.

[JM94] C. Jones and S. Maharaj. The Lego Library. distributed with Lego System, February 1994.

[LP92] Z. Luo and R. Pollack. The Lego Proof Development System: A User's Manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

[Luo90] Z. Luo. An Extended Calculus of Constructions. Technical Report CST-65-90, University of Edinburgh, July 1990.

[Mal90] G. Malcolm. Data Structures and Program Transformation. *Science of Computer Programming*, 14:255–279, 1990.

[Mee92] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

[Mee96] 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.

[MFP91] 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.

[NPS90] 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.

[Ore92] Ch.E. Ore. The Extended Calculus of Constructions (ECC) with Inductive Types. *Information and Computation*, 99, Nr. 2:231–264, 1992.

[Pau96] L.C. Paulson. A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definition. Technical report, Computer Laboratory, University of Cambridge, England, 1996.

[PM93] 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.

[Pol94] R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

[Sha96] N. Shankar. Steps Towards Mechanizing Program Transformations Using PVS. Preprint submitted to Elsevier Science, 1996.

[She93] T. Sheard. Type parametric programming. Oregon Graduate Institute of Science and Technology, Portland, OR, USA, 1993.

[Tui96] D. Tuijnman. *A Categorical Approach to Functional Programming.* PhD thesis, Universität Ulm, 1996.

[vH76] 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.

A Polynomial Datatypes in Lego

```
Module poly_datatypes Import poly_functors;

[A:Type];   (* parameter type              *)
[DT:Rep];   (* representation of datatype  *)

(* ---------------------------------------------------------------------- *)
(* Formation                                                              *)
(* ---------------------------------------------------------------------- *)

[FormationType = Rep->Type->Type];

$[T_DT : FormationType];

$[T = T_DT DT];
```

```
(* --------------------------------------------------------------------- *)
(* Introduction                                                          *)
(* --------------------------------------------------------------------- *)

[IntroductionType = [dt:Rep](FFobj dt A (T A))->T A];

$[intro : {dt:Rep}IntroductionType dt];


(* --------------------------------------------------------------------- *)
(* Elimination                                                           *)
(* --------------------------------------------------------------------- *)

(* F []         _             -> unit                        *)
(* F [rec,L]    (a1,a2,..,ak) -> C(a1) * (F L (a2,...,ak))   *)
(* F [nonrec,L] (a1,a2,..,ak) -> F L (a2,..,an)              *)


[C:(T A)->Type];

[IndHyps = [l|Sel]
  list_elim Kind ([l:Sel](Arg A (T A) l)->Type)
    ([_:Arg A (T A) noargs]unit)
    (Kind_elim
       ([a:Kind]{l:Sel}((Arg A (T A) l)->Type)->(Arg A (T A) (cons a l))->Type)
       ([l:Sel][y:(Arg A (T A) l)->Type]
        [arg:Arg A (T A) (cons rec l)]
        (prod (C (Fst arg)) (y (Snd arg))))
       ([l:Sel][y:(Arg A (T A) l)->Type]
        [arg:Arg A (T A) (cons nonrec l)]
        (y (Snd arg))))
    l];


(* IndSteps: yields product of induction steps for each constructor    *)

(* F []              _              --> unit                     *)
(* F [l1,l2,...,ln] (c1,c2,...,cn) -->                           *)
(*     IndStep1 * (F [l2,...,ln] (c2,...,cn))                    *)
(* where IndStep1 := {a:Arg A (T A) l1}(IndHyps a)->C (c1 a)   *)

[IndSteps = [DT|Rep]
   list_elim Sel ([DT:Rep](IntroductionType DT)->Type)
     ([_:empty->(T A)]unit)
     ([l:Sel][DT:Rep][y:(IntroductionType DT)->Type]
      [intro:IntroductionType (cons l DT)]
      [intro1 = compose intro (injectl l DT A (T A))]
      [intro2 = compose intro (injectr l DT A (T A))]
      (prod ({a:Arg A (T A) l}(IndHyps a)->C (intro1 a))
            (y intro2)))
     DT];

[EliminationType =
  {DT|Rep}(IndSteps (intro DT))->{x:T A}(C x)];
```

```
$[elim : EliminationType];


(* --------------------------------------------------------------------- *)
(* Equality rules                                                        *)
(* --------------------------------------------------------------------- *)

(* F []          _              -> void                              *)
(* F [rec,L]     (a1,a2,..,ak) -> ((elim fcts a1), F L (a2,...,ak))  *)
(* F [nonrec,L] (a1,a2,..,ak) -> F L (a2,..,an)                      *)


[RecCalls = [DT|Rep][fcts:IndSteps (intro DT)][l|Sel]
   list_elim Kind ([l:Sel]{a:Arg A (T A) l}(IndHyps a))
     ([_:Arg A (T A) noargs]void)
     (Kind_elim
       ([arg:Kind]{l:Sel}
          ({a:Arg A (T A) l}(IndHyps a))->
             {a:Arg A (T A) (cons arg l)}(IndHyps a))
       ([l:Sel][y:{a:Arg A (T A) l}(IndHyps a)]
        [arg:Arg A (T A) (cons rec l)]
        Pair (elim fcts (Fst arg))
             (y (Snd arg)))
       ([l:Sel][y:{a:Arg A (T A) l}(IndHyps a)]
        [arg:Arg A (T A) (cons nonrec l)]
        (y (Snd arg))))
     l];


(* F []             _                              _               --> true    *)
(* F [l1,l2,...,ln] (intro1,intro2,...,intron) (f1,f2,...,fn)  -->         *)
(*     eq1 /\ (F [l2,...,ln] (intro2,...,intron) (f2,...,fn))              *)
(*  where eq1 := (elim (f2,...,fn) (intro1 a))                             *)
(*                  == (f1 a (RecCalls (f1,f2,...,fn) a))                  *)

[Equalities = {DT|Rep}{fcts:IndSteps (intro DT)}
  list_elim Sel
    ([DT:Rep]{intro:IntroductionType DT}(IndSteps intro)->Prop)
    ([_:empty->T A][_:unit]trueProp)
    ([l:Sel][DT:Rep]
     [y:{intro:IntroductionType DT}(IndSteps intro)->Prop]
     [intro:IntroductionType (cons l DT)]
      [intro1 = compose intro (injectl l DT A (T A))]
      [intro2 = compose intro (injectr l DT A (T A))]
     [fs:IndSteps intro]
     (and ({a:Arg A (T A) l}(Eq (elim fcts (intro1 a))
                                ((Fst fs) a (RecCalls fcts a))))
           (y intro2 (Snd fs))))
    DT (intro DT) fcts];

$[eqs : Equalities];

Discharge A;
```

This article was processed using the LaTeX macro package with LLNCS style