Abstract

Formal reasoning in proof assistants, also known as mechanization, has high development costs. Building modular reusable components is a key issue in reducing these costs. A stumbling block for reuse is that inductive definitions and proofs are closed to extension. This is a manifestation of the expression problem that has been addressed by the Meta-Theory à la Carte (MTC) framework in the context of programming language meta-theory. However, MTC’s use of extensible Church-encodings is unsatisfactory.

This paper takes a better approach to the problem with datatype-generic programming (DGP). It applies well-known DGP techniques to represent modular datatypes, to build functions from functor algebras with folds and to compose proofs from proof algebras by means of induction. Moreover, for certain functionality and proofs our approach can achieve more reuse than MTC: instead of composing modular components we provide a single generic definition once and for all.

Categories and Subject Descriptors D.2.13 [Reusable Software]: Reusable libraries

General Terms Languages

Keywords datatype-generic programming, fixed points, modularity, mechanized meta-theory

1. Introduction

Meta-theory of programming languages is a core topic of computer science that concerns itself with the formalization of propositions about programming languages, their semantics and related systems like type systems. Mechanizing formal meta-theory in proof assistants is crucial, both for the increased confidence in complex de-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Generic Datatypes à la Carte

Steven Keuchel
Ghent University
steven.keuchel@ugent.be

Tom Schrijvers
Ghent University
tom.schrijvers@ugent.be

ing and reasoning about extensible modular datatypes and extensible modular functions thereby gaining modular component reuse. MTC builds on Datatypes à la Carte (DTC) [26], a Haskell solution to the expression problem, to achieve modularity. Besides writing modular algebraic datatypes for expressing semantic functions as folds, MTC also supports writing generally recursive functions using mixins and bounded fixed points. On top of that MTC presents techniques for modularly composing proofs by induction for structurally recursive functions and step-bounded induction for generally recursive functions.

The transition from the Haskell setting of DTC to a proof assistant like Coq comes with two major hurdles. DTC relies on a general fixed point combinator to define fixed points for arbitrary functors and uses a generic fold operation that is not structurally recursive. To keep logical consistency, Coq applies conservative restrictions and rejects both: a) DTC’s type level fixed-points because it cannot see that the definition is always strictly-positive, and b) DTC’s fold operator because it cannot determine termination automatically.

MTC solves both problems by using extensible Church-encodings instead. Yet, this solution leaves much to be desired.

1. By using Church-encodings MTC is forced to rely on Coq’s impredicative-set option, which is known to be inconsistent with some standard axioms of classical mathematics.

2. The fixpoint combinator provided by Church-encodings admits too many functors. For inductive reasoning, only strictly-positive functors are valid, i.e., those functors whose fixpoints are inductive datatypes. Yet, Church-encodings do not rule out other functors. Hence, in order to reason only about inductive values, MTC requires a witness of inductivity: the universal property of folds. Since every value comes with its own implementation of the fold operator, MTC needs to keep track of a different such witness for every value. It does so by decorating the value with its witness. This decoration obviously impairs the readability of the code. Moreover, since proofs are opaque in Coq, it also causes problems for equality of terms. Finally, the decoration makes it unclear whether MTC adequately encodes fixpoints.

3. Church-encodings do not support proper induction principles. MTC relies on a poor-man’s induction principle instead and requires the user to provide additional well-formedness proofs. Even though these can be automated with proof tactics, they nevertheless complicate the use of the framework.

This paper applies well-known techniques from datatype-generic programming (DGP) to overcome all of the above problems. For this purpose, it extends Schwaab and Siek’s Agda-based approach [25] to encompass all of MTC’s features in Coq.

Our specific contributions are:

- We show how to solve the expression problem in the restricted setting of Coq. We build modular datatypes, modular
functions and modular proofs from well-studied DGP representations of fixpoints for different classes of functors. In particular, we consider polynomial functors like Schwaab and Siek, but also the more expressive container types which are useful for modelling MTC’s lambda binders that are based on (parametric) higher-order abstract syntax.

- Our approach avoids impredicativity in Coq and adequately encodes fixpoints. It achieves these properties by exploiting DGP approaches that capture only strictly-positive functors.
- We show how to obtain more reuse than MTC by complementing modular definitions with generic definitions.
- We show how to apply MTC’s automatic construction of fixpoints for datatypes, relations and proofs to the DGP setting and thus improve over Schwaab and Siek’s manual fixpoint construction.

**Code and Notational Conventions** While all the code underlying this paper has been developed in Coq, the paper adopts a terser syntax for its many code fragments. For the computational parts, this syntax exactly coincides with Haskell syntax, while it is an extrapolation of Haskell syntax style for propositions and proof concepts. The Coq code is available at https://github.ugent.be/skeuchel/gdttc.

## 2. Constructions à la Carte

This section reviews the core ideas behind DTC and MTC and presents the infrastructure for writing modular functions over modular datatypes. In the next section we discuss our adapted approach that works in the restricted Coq setting.

### 2.1 Fixed points

In DTC extensible datatypes are represented as fixed points of signature functors.

```coq
data FixD f = InD { outD :: f (FixD f) }
```

For example the functors $\text{Arith}_F$ and $\text{Logic}_F$ are signatures for arithmetic and boolean expressions.

```coq
data Arith_F e = Lit Int | Add e e
data Logic_F e = BLit Bool | If e e e
```

For example $\text{Arith}_D$ is a type that features only arithmetic expressions.

```coq
type Arith_D = FixD Arith_F
```

Different features can be combined modularly by taking the coproduct of the signatures before taking the fixed point.

```coq
data (⊕) f g a = Inl (f a) | Inr (g a)
type Exp_D = FixD (Arith_D ⊕ Logic_F)
```

### 2.2 Automated injections

Combining signatures makes writing expressions difficult. For example the arithmetic expression $3 + 4$ is represented as the term

```coq
ex1 :: FixD (Arith_F ⊕ Logic_F)
ex1 = InD (Inl (Add (InD (Inl (Lit 3))) (InD (Inl (Lit 4)))))
```

Writing such expressions manually is too cumbersome and unreadable. Moreover, if we extend the datatype with a new signature other injections are needed.

To facilitate writing expressions and make reuse possible we use the sub-functor $f ∼<: g$ relation shown in Figure 1. The member function $\text{inj}$ injects the sub-functor $f$ into the super-functor $g$. In our case we need injections of functors into coproducts which are automated using type class machinery.1

```coq
instance (f ∼<: f) where
  inj = id
instance (f ∼<: g) ⇒ (f ∼<: (g ⊕ h)) where
  inj = Inl
instance (f ∼<: h) ⇒ (f ∼<: (g ⊕ h)) where
  inj = Inr
```

The $\text{inject}$ function is a variation of $\text{inj}$ that additionally applies the constructor of the fixpoint type. Using the sub-functor relation we can define smart constructors for arithmetic expressions

```coq
lit :: (Arith_F ∼<: f) ⇒ Int → FixD f
lit i = inject (Lit i)
add :: (Arith_F ∼<: f) ⇒ FixD f → FixD f → FixD f
add a b = inject (Add a b)
```

that construct terms of any abstract super-functor $f$ of $\text{Arith}_F$. This is essential for modularity and reuse. We can define terms using the smart-constructors, but constructing a value of a specific fixpoint datatype is delayed. With these smart constructors the above example term becomes

```coq
ex1′ :: (Arith_F ∼<: f) ⇒ FixD f
ex1′ = lit 3’add’ lit 4
```

The $\text{prj}$ member function is a partial inverse of $\text{inj}$. With it we can test if a specific sub-functor was used to build the top layer of a value. This operation fails if another sub-functor was used. The type class also includes proofs that witness the partial inversion. The $\text{project}$ function is a variation of $\text{prj}$ that strips the constructor of the fixpoint type. Similarly to injections, we can automate projections for coproducts by adding corresponding definitions to the instances above.

### 2.3 Semantic functions

In this section we define evaluation for arithmetic and boolean expressions modularly. We use another modular datatype to represent values. Its signatures and smart-constructors are given in Figure 2. The signature $\text{Stuck Value}_F$ represents a sentinel value to signal type errors during evaluation.

If $f$ is a functor, we can fold over any value of type $\text{FixD}_D f$ as follows:

```coq
class f ∼<: g where
  inj :: f a → g a
  prj :: g a → Maybe (f a)
  inj_prj :: ∀ a (ga :: g a) (fa :: f a).
  prj ga = Just fa → ga = inj fa
  prj_prj :: ∀ a (fa :: f a).
  prj (inj fa) = Just fa
inject :: (f ∼<: g) ⇒ f (FixD_D g) → FixD_D g
inject x = FixD_D $ inj x
project :: (f ∼<: g) ⇒ FixD_D g → Maybe (f (FixD_D g))
project x = prj $ unFix x
```

Figure 1. Sub-functor relation

---

1 Coq’s type-class mechanism performs backtracking. These instances do not properly work in Haskell. See [26] for a partial solution.
Function algebras for different signatures can be combined together to get an algebra for their coproduct. The necessary instance declaration is also given in Figure 3. Finally, we can define an evaluation function for terms given an FAlgebra instance for Eval.

$$[] :: (\text{Functor } f, \text{FAlgebra Eval } f (\text{Fix}_D \text{vf})) \Rightarrow \text{Fix}_D f \rightarrow \text{Fix}_D \text{vf}$$
$$[] = \text{fold}_D (f \text{.algebra Eval})$$

3. Reasoning à la Carte

In Section 2 we focused on programming in Haskell. In this Section we turn our attention towards performing modular constructions of datatypes, functions and inductive proofs in a proof-assistant like Coq.

3.1 Modular Definitions in Coq

Unfortunately, we cannot directly translate the definitions of Section 2. Coq requires all inductive definitions to be strictly-positive. We define strictly positive types (SPT) by using the following generative grammar [1]:

$$\tau ::= X \mid 0 \mid 1 \mid \tau + \tau \mid \tau \times \tau \mid K \rightarrow \tau \mid \mu X.\tau$$

where X ranges over type variables and K ranges over constant types, i.e. an SPT with no free type variables. The constants 0 and 1 represent the empty and unit types, the operators +, ×, → represent coproduct, cartesian product, exponentiation and least fixed point construction.

For Fix\_D f to be strictly positive this means that the argument functor f has to be strictly-positive, i.e. it corresponds to a term built with the above grammar with one free type variable.

As a counter example, inlining the non-strictly positive functor X ↦ (X → Int) → Int into Fix\_D yields the invalid datatype definition

$$\text{data NSP} = \text{NSP} ((\text{NSP} \rightarrow \text{Int}) \rightarrow \text{Int})$$

It does not satisfy the positivity requirements and is rejected by Coq. While Coq can automatically determine the positivity for any concrete functor by inspecting its definition, it cannot do so for an abstract functor like the one that appears in the definition of Fix\_D. Hence, Coq also rejects Fix\_D.

Of course, we have no intention of using non-strictly positive functors for our application and would like to provide the evidence of strict-positivity to the fixpoint type constructor. Mini-Agda [2] for example allows programmers to annotate strictly-positive and negative positions of type constructors. Unfortunately, Coq does not provide us with this possibility and a different approach is needed. To this end, we define the SFP type class in Figure 4 which serves as a declarative specification of our requirements on functors and which carries the required evidence.

While we need the existence of a fixed point type of abstract super-functors, it is inessential how it is constructed. This means that instead of providing a generic fixpoint type constructor like Fix\_D we can alternatively provide a witness of the existence of a
valid fixpoint in the type class, i.e. we make the fixpoint an associated type of the SPF type class. We thereby delay the problem of defining it to the specific functors. SPF also includes the functions \( \text{in}_F \) and \( \text{out}_F \) as members that fold/unfold one layer of the fixpoint.

The fold operator from Section 2 also causes problems in Coq. For any concrete functor we can inline the definition of \( \text{fmap} \) to get \( \text{fold}_f = \text{fold}_{\text{fmap}} \text{fold}_f \) pass the termination check, but again we are working with an abstract functor \( f \) and an abstract functorial mapping \( \text{fmap} \). Similarly, we resolve this by including a witness for the existence of a valid fold operator in the SPF class.

### 3.2 Modular inductive reasoning

The SPF typeclass also provides an interface for reasoning. It includes proof terms that witness that folding/unfolding of the fixpoint type form inverse operations and that the provided fold operators satisfies the universal property of folds. The last missing piece for reasoning is to have an induction principle available.

Consider the induction principle \( \text{ind}_A \) for arithmetic expression.

\[
\text{ind}_A :: \forall p :: (\text{Arith} \rightarrow \text{Prop}). \quad \\
\forall h :: (\forall n. \quad p (\text{in}_F (\text{Lit} n))). \quad \\
\forall h :: (\forall x. y. p x \rightarrow p y \rightarrow p (\text{in}_F (\text{Add} x y))). \quad \\
\forall x. p x
\]

It takes a proposition \( p \) as parameter and inductive steps \( h \) and \( h_a \) for each case of the initial algebra. We say that \( h \) and \( h_a \) together form a proof algebra of \( p \). An inductive step consists of showing \( p \) for an application of the initial algebra given proofs of \( p \) for all recursive positions. In case of a literal we have no recursive positions and in case of addition we have two. Proof algebras for other datatypes differ in the number of cases and the number of recursive positions.

In the following we develop a uniform representation of proof algebras to allow their modularization. We use an all modality [7] for functors to capture the proofs of recursive positions. Informally, the all modality of a functor \( f \) and a predicate \( p :: a \rightarrow Prop \) is a new type \( \text{All} a p :: f a \rightarrow Prop \) that says that the predicate \( p \) holds for each \( x :: a \) in an \( a \). The following type \( \text{ArithAll} \) is an example of an all modality for arithmetic expressions.

\[
\text{data ArithAll} a p :: \text{Arith}_F a \rightarrow Prop \text{ where} \\
\text{ALit} :: \\
\text{AAdd} :: p x \rightarrow p y \rightarrow \text{ArithAll} a p (\text{Add} x y)
\]

We introduce a new typeclass \( \text{PFunctor} \) that carries the associated all modality type and make SPF a subclass of it. Using the all modality definition we can write \( \text{ind}_A \) equivalently as:

\[
\text{ind}_A :: \forall p :: (\text{Arith} \rightarrow \text{Prop}). \quad \\
\forall h :: (\forall x. \text{ArithAll} a p x \rightarrow p (\text{in}_F x)). \quad \\
\forall x. p x
\]

The proof algebra is now a single parameter \( h \). Note that \( h \) shows that \( p \) holds for an application of the initial algebra \( \text{in}_F \).

In the modular setting however, we only want to provide proofs for sub-algebras of the initial algebra that correspond to specific signatures and combine these proof sub-algebras to a complete proof algebra for the initial algebra. To this end, we define proof algebras in Figure 4 more generally over arbitrary algebras. As a last member of SPF we introduce \( \text{ind} \) that is an induction principle for the fixpoint type \( \text{Fix}_F \). It takes a proof algebra of a property \( p \) for the initial algebra and constructs a proof for every value of \( \text{Fix}_F \).

### 3.3 Composing features

In Section 2 we have shown how to modularly compose signatures and semantic functions. These definitions carry over to Coq without any problems. We now turn to modularly composing proofs.

The \( \text{PFunctor} \) class also has the nice property of being closed under coproducts.

\[
\text{instance} (\text{PFunctor} f, \text{PFunctor} g) \Rightarrow \text{PFunctor} (f \oplus g) \text{ where} \\
\text{type} \quad \text{All} a p x s :: \text{case} x s\text{ of} \\
\text{Inl ks} \rightarrow \text{All} a p x s \\
\text{Inr ks} \rightarrow \text{All} a p x s
\]

As for function algebras, we can use a type class to define and assemble proof algebras in a modular fashion.

\[
\text{class} \quad (\text{ProofAlgebra} f a alg p) \Rightarrow \text{ProofAlgebra} (f \oplus a) p \text{ where} \\
\text{palgebra} :: \text{PA::Functor} f a alg p
\]

\[
\text{instance} (\text{ProofAlgebra} f a alg p) \Rightarrow \text{ProofAlgebra} (f \oplus a) p \text{ where} \\
\text{palgebra} :: \text{PA::Functor} f a alg p
\]

When instantiating modular functions to a specific set of signatures, we need an SPF instance for the coproduct of that set. As with algebras we would like to derive an instance for \( f \oplus g \) given instances for \( f \) and \( g \) as we cannot expect the programmer to provide an instance for every possible set of signatures. Unfortunately, SPF does not include enough information about the functors to do
this in a constructive way. What we need is a refinement of SPF that allows us to perform this construction.

4. Containers

This paper uses techniques from datatype-generic programming (DGP) to get a compositional refinement of SPF. The problem of defining fixpoints for a class of functors also arises in many approaches to DGP and we can use the same techniques in our setting.

In a dependently-typed setting it is common to use a universe for generic programming [4, 7]. A universe consists of two important parts:

1. A set Code of codes that represent types in the universe.
2. An interpretation function Ext that maps codes to types.

There is a large number of approaches to DGP that vary in the class of types they can represent and the generic functions they admit. For our application we choose the universe of containers [1]. In this section we review containers for generic programming and show how to resolve the problem of implementing folds and induction in a modular way by using generic implementations.

An important property of this universe is that all strictly-positive functors can be represented as containers [1]. In this respect we do not loose any expressivity.

4.1 Universe

The codes of the container universe are of the form $S \triangleleft P$ where $S$ denotes a type of shapes and $P : S \rightarrow *$ denotes a family of position types indexed by $S$. The extension $Ext$ $c$ $a$ of a container $c$ in Figure 5 is a functor. A value of the extensions $Ext$ $c$ $a$ consists of a shape $s : shape$ $c$ and for each position $p : pos$ $c$ $s$ of the given shape we have a value of type $a$. We can define the functorial mapping $g$ $map$ generically for any container.

$g$ $map$ $(a \rightarrow b) \rightarrow Ext$ $c$ $a$ $\rightarrow Ext$ $c$ $b$

$g$ $map$ $f$ $(Ext$ $s$ $pf)$ $= Ext$ $s$ $(\lambda p \rightarrow f$ $(pf$ $p))$

The functor $Arith_c$ for arithmetic expressions can be represented as a container functor using the following shape and position type.

data $Arith_S$ $=$ $Lit_S$ $Int$ $|$ $Add_S$ data $Arith_P$ $::$ $Arith_S$ $\rightarrow$ $\ast$ where $Add_{p_1}$ $::$ $Arith_P$ $Add_S$

$Add_{p_2}$ $::$ $Arith_P$ $Add_S$

type $Arith_C$ $::$ $Arith_S$ $\triangleright$ $Arith_P$

The shape of an $Arith_c$ value is either a literal $Lit$ with some integer value or it is an addition $Add$. In case of $Add$ we have two recursive positions $Add_{p_1}$ and $Add_{p_2}$. $Lit$ does not have any recursive positions.

The isomorphism between $Arith_c$ and $Ext$ $Arith_C$ is witnessed by the following two conversion functions.

4.2 Fixpoints and folds

The universe of containers allows multiple generic constructions. First of all, the fixpoint of a container is given by its W-type.

data $W$ $(c : Cont)$ $::$ $\{\ unSup :: Ext$ $c$ $(W$ $c) \}$

The definition of $Ext$ is known at this point and Coq can see that the $W$ $c$ is strictly positive for any container $c$ and hence the definition of $W$ is accepted.

Furthermore, we define a fold operator generically.

$g$ $fold$ $::$ $\{\ Algebra$ $(Ext$ $c$ $a$ $\rightarrow$ $W$ $c$ $\rightarrow$ $a)$

$g$ $fold$ $alg$ $(Sup$ $(Ext$ $s$ $pf))$ $= alg$ $(Ext$ $s$ $(\lambda p \rightarrow g$ $fold$ $alg$ $(pf$ $p)))$

Note that this definition is essentially the same as the definition of $fold_D$ from Section 2. Because of the generic implementation of $g$ $map$ we can inline it to expose the structural recursion. Coq accepts this definition, since the recursive call $g$ $fold$ $alg$ $(pf$ $p)$ is performed on the structurally smaller argument $pf$ $p$.

4.3 Coproducts

Given two containers $S_1 \triangleright P_1$ and $S_2 \triangleright P_2$ we can construct a coproduct. The shape of the coproduct is given by the coproducts of the shape and the family of position types delegates the shape to the families $P_1$ and $P_2$.

type $S_+ = Either$ $S_1$ $S_2$

type $P_+ (Left$ $s) = P_1$ $s$

type $P_+ (Right$ $s) = P_2$ $s$

The injection functions on the extensions are given by

$inl :: Ext$ $(S_1 \triangleright P_1)$ $\rightarrow Ext$ $(S_+ \triangleright P_+)$

$inl (Ext$ $s$ $pf)$ $= Ext$ $(Left$ $s)$ $pf$

$inr :: Ext$ $(S_2 \triangleright P_2)$ $\rightarrow Ext$ $(S_+ \triangleright P_+)$

$inr (Ext$ $s$ $pf)$ $= Ext$ $(Right$ $s)$ $pf$

4.4 Induction

To define an induction principle for container types we proceed in the same way as in Section 3.2 by defining proof algebras using an
class IFunctor i (f :: (i → Prop) → i → Prop) where
  fmap :: ∀(a :: i → Prop) (b :: i → Prop) (j :: i). (v j . a j → b j) → f a j → f b j

class IFunctor i f ⇒ ISPF i (f :: (i → Prop) → i → Prop) where
  instance ISPF i p ⇒ ISPF i (p ∘ f)

type IFix i f :: (i → Prop) ⇒ Prop

data AWS p :: (Prop → Prop) a where
  AWS :: ∀(a :: Prop) (p :: Prop → Prop) → Prop

defining the type class Container and giving instances for
the definition of the extension and the fixed point of an indexed
functor. Similarly to containers, one can generically define a fold
operator for all indexed containers and construct the coproduct of
two indexed containers.

4.6 Extensible logical relations

Many properties are expressed as logical relations over datatypes.
These relations are represented by inductive families where a con-
structor of the family corresponds to a rule defining the relation.

When using logical relations over extensible datatypes the set of
rules must be extensible as well.

Extensibility of inductive families is obtained in the same way
as for inductive datatypes by modularly building inductive families
as fixpoints of functors between inductive families. The following
indexed functor \(WTNat\) covers the rule that a natural number
value has a natural number type.

data WTNat (wv :: (FixP pf, FixP tf) → Prop) :: (Value, Type) → Prop where
  WTNat :: (NatValueF ≪ v, NatTypeF ≪ tf) ⇒ WTNat wv (wi n, tnat)

MTC constructs fixed points of indexed functors also by means
of Church-encodings. The indexed variants of algebras and fixed
points are

\[\text{type IAlgebra i f a} = \forall (j :: i). f a j \rightarrow a j\]

For type-soundness proofs we perform folds over proof-terms in
order to establish propositions on the indices and hence make use
of the fold operation provided by Church-encodings. However,
contrary to inductive datatypes we do not make use of propositions
on proof-terms and hence do not need an induction principle for
them. This also means that we do not need to keep track of the
universal property of folds for proof-terms. Figure 7 defines the
type class ISPF that collects the necessary functions for modularly
building logical relations.

Alternatively, we can use a universe of indexed containers [3].
An i-indexed container is essentially a container together with an
assignment of indices for each shape and each position of that
shape.

More formally, an i-indexed container \(S \mathrel{\triangleright} P \mathrel{\triangleright} R\) is given by
a family of shapes \(S :: i \rightarrow \ast\) and family of position types
\(P :: (j :: i) \rightarrow S j \rightarrow \ast\) and an assignment \(R :: (j :: i) \rightarrow (s :: S j) \rightarrow P j s \rightarrow i\) of indices for positions. Figure 8 gives the
definition of the extension and the fixed point of an indexed
container. Similarly to containers, one can generically define a fold
operator for all indexed containers and construct the coproduct of
two indexed containers.

Fixed points and fold operators can be defined generically on
that universe similarly to Section 4.2. Indexed containers are also
closed under coproducts and indexed algebras can be modularly
composed using type classes.

5. Polynomial functors

In the previous section we have implemented generic functions for
functorial mappings, fixed points, folds and generic proofs about
their properties.

Other common functionality can be treated with generic imple-
mentations as well. In this section we look at a generic implement-
ation of equality testing and proofs about its correctness. These
functions are used for example in the MTC framework in the imple-
mentation of a modular type-checker that tests if both branches of
an if expression have the same type and that the function and argument
type of a function application are compatible. Furthermore for
reasoning about functions that use equality testing we need proofs

Figure 7. Indexed strictly-positive functor class

all modality [7]. The all modality on containers is given generically
by a \(\Pi\)-type that asserts that \(q\) holds at all positions.

\[\text{GAll} :: (q :: a \rightarrow Prop) \rightarrow \text{Ext c a} \rightarrow Prop\]

\[\text{GAll} q (\text{Ext s pf}) = \forall (p :: \text{pos} s c). q (p f p)\]

As with the implementation of the generic fold operations, enough
structure is exposed to write a valid induction function: \(\text{gind}\) calls
itself recursively on the structurally smaller values \(pf p\) to establish
the proofs of the recursive positions before applying the proof
algebra \(\text{palg}\).

\[\text{gind} :: \forall (c :: \text{Cont}) \rightarrow\]

\[\forall q :: \forall \text{c w Prop} \rightarrow\]

\[\forall (\text{palg} :: \forall x\text{.GAll q xs} \rightarrow q (\text{Sup xs})) \rightarrow\]

\[\forall x. q x\]

\[\text{gind c q palg} (\text{Sup} (\text{Ext s pf})) =\]

\[\text{palg} (\lambda p \rightarrow \text{gind c q palg} (p f p))\]

4.5 Container functor class

Directly working with the container representation is cumbersome
for the user. As a syntactic convenience we allow the user to use any
conventional functor of type \(\ast \rightarrow \ast\) as long as it is isomorphic to
a container functor. The type class \(\text{Container}\) in Figure 6 witnesses
this isomorphism. The class contains the functions \(\text{from}\) and \(\text{to}\)
that perform the conversion between a conventional functor and a
container functor and proofs that these conversions are inverses.

Via the isomorphisms \(\text{from}\) and \(\text{to}\) we can import all the generic
functions to concrete functors and give instances for \(\text{Functor, PFunctor}\)
and \(\text{SPF}\).

instance Container f ⇒ Functor f where
  fmap f = to o g f o from

instance Container f ⇒ PFunctor f where
  All Q = GAll Q o from

instance Container f ⇒ SPF f where
  FixF = W S P
  inf = sup o from
  outF = to o unSup
  fold alg = gfold (alg o to)

The important difference to the \(\text{SPF}\) class is that we can generi-
cally build the instance for the coproduct of two \(\text{Container}\)
funtors by using the coproduct of their containers.

instance (Container f, Container g) ⇒ Container (f ⊕ g)
5.1 Universe

The codes Poly and interpretation Ext_P of the polynomial functor universe are shown in Figure 10. A polynomial functor is either the constant unit functor U, the identity functor I, a coproduct C p1 p2 of two functors, or the cartesian product P p1 p2 of two functors. The interpretation Ext_P is defined as an inductive family indexed by the codes.

As an example consider the functor FunType that can represent function types of an object language.

data FunType a = TArrow a a

It has a single constructor with two recursive positions for the domain and range types. Hence it can be represented by the code P I I. The conversion functions between the generic and conventional representation are given by

fromFunType :: FunType a → Ext_P (P I I) a

fromFunType (TArrow x y) = EP (EI x) (EI y)

toFunType :: Ext_P (P I I) a → FunType a
toFunType (EP (EI x) (EI y)) = TArrow x y

As before we define a type-class Polynomial that carries the conversion functions and isomorphism proofs. The definition of the class is also given in Figure 10. An instance for FunType is the following, with proofs omitted:

instance Polynomial FunType where
    pcode = P I I
    pto = toFunType
    pfrom = fromFunType
    pfromToInverse = ...
    pfromToInverse = ...

5.2 Universe embedding

To write modular functions for polynomial functors we proceed in the same way as in Section 4 by showing that Polynomial is closed under coproducts and building the functionality of the SPF type class generically.

However, that would duplicate the generic functionality and would prevent us from using polynomial functors with containers. Since containers are closed under products and coproducts we can embed the universe of polynomial functors in the universe of containers. In order to do this, we have to derive a shape type from the code of a polynomial functor and a family of position types for each shape. We can compute the shape by recurusing over the code.

polyC :: Poly → *
ipoly U = ()
ipoly I = ()
ipoly (C c d) = polyC c + polyC d
ipoly (P c d) = (polyC c, polyC d)

The constant unit functor and the identity functor have only one shape which is represented by a unit type. As in section 4.3 the shape of a coproduct is the coproduct of the shapes of the summands and the shape of a product is the product of shapes of the factors. The definition of positions also proceeds by recurusing over the code.

data Poly = U | I | C Poly Poly | P Poly Poly

data Ext_P (c :: Poly) (a :: *) where
    EU :: Ext_P U a
    EI :: a → Ext_P I a
    EL :: Ext_P C c d → Ext_P (C c d) a
    ER :: Ext_P d a → Ext_P (C c d) a
    EP :: Ext_P c a → Ext_P (P c d) a

class Polynomial f where
    pcode :: Poly
    pto :: Ext_P pcode a → f a
    pfrom :: f a → Ext_P pcode a
    pfromToInverse :: ∀a.pto (pfrom a) = a
    pfromToInverse :: ∀a.pfrom (pto a) = a
The constant unit functor does not have any positions and the identity functor has exactly one position. For coproducts the positions are the same as the ones of the chosen summand and for a product we take the disjoint union of the positions of the shapes of the components.

The next essential piece for completing the universe embedding are conversions between the interpretations of the codes. The function \( ptU \) converts the polynomial interpretation to the container interpretation.

\[
ptU :: (c :: Poly) \to \text{Ext}_c c a \to \text{Ext} (\text{polys}_c \triangleright \text{poly}_c c) a
\]

\[
\begin{align*}
\text{ptU}_U &= \text{Ext} () (\lambda p \to \text{case } p \text{ of }) & \text{ptU}_I &=(\text{El } a) = \text{Ext } () (\lambda () \to a) \\
\text{ptU}_{(C c d)} (\text{EL } x) & = \text{Ext} (\text{Left } s) pf
\end{align*}
\]

\[\text{where } \text{Ext } s \text{ pf } = \text{ptU}_n c x\]

\[\text{ptU}_{(C c d)} (\text{ER } y) = \text{Ext} (\text{Right } s) pf\]

\[\text{where } \text{Ext } s_1 \text{ pf } = \text{ptU}_n c y \quad \text{Ext } s_2 \text{ pf } = \text{ptU}_n d y\]

Similarly we define the function \( pf\) that performs the conversion in the opposite direction. We omit the implementation.

\[
pf :: (c :: Poly) \to \text{Ext} (\text{polys}_c \triangleright \text{poly}_c c) a \to \text{Ext}_c c a
\]

To transport properties across these conversion functions we need the implementation.\( ptU \) from Unary and \( pf \) from Unary convert the polynomial interpretation to the container interpretation.

\[
\begin{align*}
\text{Inst}(\text{poly}_c c & \triangleright \text{poly}_c c) \to \text{poly}_c c \\
\text{Inst}(\text{poly}_c c & \triangleright \text{poly}_c c) \to \text{poly}_c c
\end{align*}
\]

In the same vein we can prove the properties of the \( Eq \) type class for this equality function using mutual induction over fixed points and partially constructed fixed points.

\[
\begin{align*}
go \ (C c d) (\text{EL } x) (\text{ER } y) &= \text{False} \\
go \ (C c d) (\text{ER } x) (\text{EL } y) &= \text{False} \\
go \ (C c d) (\text{ER } x) (\text{ER } y) &= \text{go } d \ x \ y \\
go \ (P c d) (\text{EP } x y) &= \text{go } c \ x \ y \land \text{go } d \ x \ y
\end{align*}
\]

\[
\text{In this section we discuss the benefits and trade-offs we have experienced while porting the case study to our approach.}
\]

\[\text{Code size} \quad \text{By the move to a datatype-generic approach the underlying modular framework grew from about 2500 LoC to about 3500 LoC. This includes both the universe of containers and polynomial functors and the generic implementations of fold, induction and equality.}\]

\[\text{The size of the implementation of the modular feature components is roughly 1100 LoC per feature in the original MTC case study. By switching from Church-encodings to a datatype-generic approach we stripped away on average 70 LoC of additional proof obligations needed for reasoning with Church-encodings per feature. However, instatiating the MTC interface amounts to roughly 40 LoC per feature while our approach requires about 100 LoC per feature for the instances.}\]

\[\text{By using the generic equality and generic proofs about its properties we can remove the specific implementations from the case study. This is about 40 LoC per feature. In total we could reduce the average size of a feature implementation to 1050 LoC.}\]

\[\text{2 of step-bounded evaluation functions}\]
**Impredicative sets**  The higher-rank type in the definition of $\text{Fix}_M$

$$\text{Fix}_M \ (f :: A ightarrow B) = \forall (a :: A). \text{Algebra} f a \rightarrow a$$

causes $\text{Fix}_M$ to be in a higher universe level than the domain of $f$. Hence to use $\text{Fix}_M$ as a fixpoint of $f$ we need an impredicative sort. MTC uses Coq’s impredicative-set option for this which is known to lead to logical inconsistencies.

When constructing the fixpoint of a container we do not need to raise the universe level and can thus avoid using impredicative sets.

**Induction principles**  Church encodings have problems supporting proper induction principles, like the induction principle for arithmetic expressions $\text{ind}_A$ in Section 3.2. MTC uses a poor-man’s induction principle $\text{ind}_A^2$ instead.

$$\text{ind}_A^2 ::= \forall (p :: (\text{Arith} ightarrow \text{Prop})). \forall (hl :: (\forall n.p \ (\text{InMTC} \ (\text{Lit} \ n))). \forall (ha :: (\forall x.y.p \ x \rightarrow p \ y \rightarrow p \ (\text{InMTC} \ (\text{Add} \ x \ y)))).$$

Algebra $\text{Arith}_f (\exists a.p a)$

The induction principle uses a dependent sum type to turn a proof algebras into a regular algebra. The algebra builds a copy of the original term and a proof that the property holds for the copy. The proof for the copy can be obtained by folding with this algebra. In order to draw conclusions about the original term two additional well-formedness conditions have to be proven.

1. The proof-algebra has to be well-formed in the sense that it really builds a copy of the original term instead of producing an arbitrary term. This proof needs to be done only once for every induction principle of every functor and is about 20 LoC per feature. The use of this well-formedness proof is completely automated using type-class and hence hidden from the user.

2. The fold operator used to build the proof using the algebra needs to be a proper fold operator, i.e. it needs to satisfy the universal property of folds.

$$\text{foldMTC} :: \text{Algebra} f a \rightarrow \text{Fix}_M f \rightarrow a$$

$$\text{foldMTC} \ \text{alg} \ fa = fa \ \text{alg}$$

**Equality of terms**  Packaging universal properties with terms obfuscates equality of terms when using Church-encodings. The proof component may differ for the same underlying term.

This shows up for example in type-soundness proofs in MTC. An extensible logical relation $\text{WTValue} \ (v, t)$ is used to represent well-typing of values. The judgement ranges over values and types. The universal properties are needed for inversion lemmas and thus the judgement needs to range over the variants that are packaged with the universal properties.

However, knowing that $\text{WTValue} \ (v, t)$ and $\text{proj1} \ t = \text{proj1} \ t'$ does not directly imply $\text{WTValue} \ (v, t')$ because of the possibly distinct proof component. To solve this situation auxiliary lemmas are needed that establish the implication. Other logical relations need similar lemmas. Every feature that introduces new rules to the judgements must also provide proof algebras for these lemmas.

In the case study two logical relations need this kind of auxiliary lemmas: the relation for well-typing and a sub-value relation for continuity. Both of these relations are indexed by two modular types and hence need two lemmas each. The proofs of these four lemmas, the declaration of abstract proof algebras and the use of the lemmas amounts to roughly 30 LoC per feature.

In our approach we never package proofs together with terms and hence this problem never appears. We thereby gain better readability of proofs and a small reduction in code size.

**Adequacy**  Adequacy of definitions is an important problem in mechanizations. One concern is the adequate encoding of fixpoints. MTC relies on a side-condition to show that for a given function $f$ the folding $\text{InMTC} : f \ (\text{Fix}_M f) \rightarrow \text{Fix}_M f$ and unfolding $\text{outMTC} : \text{Fix}_M f \rightarrow f \ (\text{Fix}_M f)$ are inverse operations, namely, that all appearing $\text{Fix}_M f$ values need to have the universal property of folds. This side-condition raises the question if $\text{Fix}_M f$ is an adequate fixpoint of $f$. The pairing of terms together with their proofs of the universal property do not form a proper fixpoint either, because of the possibility of different proof components for the same underlying terms.

Our approach solve this adequacy issue. The $\text{SPF}$ type class from Figure 4 requires that $\text{in}$ and $\text{out}$ are inverse operations without any side conditions on the values and containers give rise to proper $\text{SPF}$ instances.

### 7. Related and Future Work

**DGP in proof-assistants**  Datatype-generic programming started out as a form of language extension such as Polyp [12] or Generic Haskell [17]. Yet Haskell has turned out to be powerful enough to implement datatype-generic programming in the language itself and over the time a vast number of DGP libraries for Haskell have been proposed [8, 10, 14, 19, 21, 24, 29]. Compared with a language extension, a library is much easier to develop and maintain.

Using the flexibility of dependent-types there are multiple proposals for performing datatype-generic programming in proof assistants [3, 5, 7, 18, 27]. This setting not only allows the implementation of generic functions, but also of generic proofs. The approaches vary in terms of how strictly they follow the positivity or termination restrictions imposed by the proof assistant. Some circumvent the type-checker at various points to simplify the development or presentation while others put more effort in convincing the type-checker and termination checker of the validity [23]. However, in all of the proposals there does not seem to be any fundamental problem caused by the restrictions.

**DGP for modular proofs**  Modularly composing semantics and proofs about the semantics has also been addressed by [25] in the context of programming language meta-theory. They perform their development in Agda and similar to our approach they also use a universe approach based on polynomial functors to represent modular datatypes. They split relations for small-step operational semantics and well-typing on a feature basis. However, the final fixed points are constructed manually instead of having a generic representation of inductive families.

Using Coq’s type classes both MTC and our approach also allow for more automation in the final composition of datatypes, functions and proofs. Agda features instance arguments that can
be used to replace type classes in various cases. However, the current implementation does not perform recursive resolution and as a result Agda does not support automation of the composition to the extent that is needed for DTC-like approaches.

**Combining different DGP approaches** We have shown an embedding of the universe of polynomial functors into the universe of containers. Similar inclusions between universes have been done in the literature [22]. Magalhães and Lõh [20] have ported several popular DGP approaches from Haskell to Agda and performed a formal comparison by proving inclusion relations between the approaches.

DGP approaches differ in terms of the class of datatypes they capture and the set of generic functions that can be implemented for them. Generic functions can be transported from a universe into a sub-universe. However, we are not aware of any DGP library with a systematic treatment of universes where each generic function is defined on the most general universe that supports that function.

**DGP for abstract syntax** We have shown how to obtain more reuse by complementing modular definitions with a generic equality function and generic proofs of its properties. Of course more generic functionality like traversals, pretty-printing, parsing etc. can be covered by means of datatype-generic programming

One very interesting idea is to use datatype-generic programming to handle variable binding [9, 28]. Variable binding is an ubiquitous aspect of programming languages. Moreover, a lot of functionality like variable substitutions and free variable calculations is defined for many languages. Licata and Harper [16] and Keuchel and Jeuring [13] define universes for datatypes with binding in Agda. Lee et al. [15] develop a framework for first-order representations of variable binding in Coq that is based on the universe of regular tree types [23] and that provides many of the so-called *infrastructure lemmas* required when mechanizing programming language meta-theory.

An interesting direction for future work is to extend these approaches to capture variable binding in the indices of relations on abstract syntax and use this as the underlying representation of extensible datatypes and extensible logical relations and thereby complementing modular functions with generic proofs about variable binding.

**Automatic derivation of instances** Most, if not all, generic programming libraries in Haskell use Template Haskell to derive the generic representation of user-defined types and to derive the conversion functions between them.

The GMeta [15] framework includes a standalone tool that also performs this derivation for Coq. Similarly we also like to be able to derive instances for the *Container* and *Polynomial* classes automatically.

### 8. Conclusion

Formally mechanizing proofs can be very tedious and the amount of work required for larger developments is excruciating. Meta-Theory à la Carte is a framework for modular reusable components for use in mechanizations. It builds on the Datatypes à la Carte approach to solve an extended version of the expression problem. MTC allows modular definitions of datatypes, semantic functions and logical relations and furthermore modular inductive proofs.

MTC uses extensible Church-encodings to overcome conservative restrictions imposed by the Coq proof-assistant. This approach has shortcomings in terms of confidence in the definitions and in terms of usability. This paper addresses these shortcomings by using datatype-generic programming techniques to replace Church-encodings as the underlying representation of type-level fixed points. Our approach avoids impredicativity, adequately encoding fixed points and leads to stronger induction principles by exploiting DGP approaches that capture only strictly-positive functions.

Working with generic structure representation has the added benefit that we can implement generic functions like equality and generic proofs once and for all.

### Acknowledgments

We are grateful to the anonymous reviewers for their helpful comments.

### References


design and implementation, TLDI ’03, pages 26–37, New York, NY, USA, 2003. ACM.


