Reusability for mechanized meta-theory

Steven Keuchel

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Supervisor:
prof. dr. ir. Tom Schrijvers

Department of Applied Mathematics, Computer Science and Statistics
Faculty of Sciences, Ghent University

Acknowledgments

Contents

Chapter 1

Introduction

Part I

Modularity

Chapter 2

Background

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.

2.1 Modular definitions

2.2 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 designs and as a basis for technologies such as proof-carrying code.

The PoplMark challenge [**?**] identified *component reuse* as one of several key issues of formal mechanization of programming language meta-theory. Reuse is crucial because formalizations have high costs. Unfortunately the current practice to achieve reuse is to copy an existing formalization and to change the existing definitions manually to integrate new features and to subsequently *patch up the proofs* to cater for the changes.

The recent work on *Meta-Theory à la Carte* (MTC) [**?**] is the first to improve this situation. It is a Coq framework for defining and reasoning about extensible modular datatypes and extensible modular functions thereby gaining modular component reuse. MTC builds on Datatypes à la Carte (DTC) [**?**], a Haskell solution to the expression problem, to achieve modularity. Besides writing modular algebras 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 [**?**] 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/gdtc`.

2.3 Expression problem

2.4 Datatypes à la Carte

2.5 Church encodings

2.6 Metatheory à la Carte

Chapter 3

Modular predicative universes

3.1 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.

3.1.1 Fixed points

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

$$\textbf{data}\ Fix_D\ f = In_D\ \{\, out_D :: f\ (Fix_D\ f)\,\}$$

For example the functors $Arith_F$ and $Logic_F$ are signatures for arithmetic and boolean expressions.

$$\begin{array}{l}\textbf{data}\ Arith_F\ e = Lit\ Int \quad \mid Add\ e\ e\\ \textbf{data}\ Logic_F\ e = BLit\ Bool \mid If\ e\ e\ e\end{array}$$

For example $Arith_D$ is a type that features only arithmetic expressions.

$$\textbf{type}\ Arith_D = Fix_D\ Arith_F$$

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

```
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_inj  :: ∀a (fa :: f a).
    prj (inj fa) = Just fa
inject :: (f ≺: g) ⇒ f (Fix_D g) → Fix_D g
inject x = Fix_D $ inj x
project :: (f ≺: g) ⇒ Fix_D g → Maybe (f (Fix_D g))
project x = prj $ unFix x
```

Figure 3.1: Sub-functor relation

```
data (⊕) f g a = Inl (f a) | Inr (g a)
type Exp_D = Fix_D (Arith_F ⊕ Logic_F)
```

3.1.2 Automated injections

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

```
ex1 :: Fix_D (Arith_F ⊕ Logic_F)
ex1 = In_D (Inl (Add
                  (In_D (Inl (Lit 3)))
                  (In_D (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 $f \prec: g$ relation shown in Figure 3.1. The member function *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]

[1] Coq's type-class mechanism performs backtracking. These instances do not properly work in Haskell. See [**?**] for a partial solution.

```
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 *inject* function is a variation of *inj* that additionally applies the constructor of the fixpoint type. Using the sub-functor relation we can define smart constructors for arithmetic expressions

```
lit :: (Arith_F ≺: f) ⇒ Int → Fix_D f
lit i = inject (Lit i)
add :: (Arith_F ≺: f) ⇒ Fix_D f → Fix_D f → Fix_D f
add a b = inject (Add a b)
```

that construct terms of any abstract super-functor f of $Arith_F$. 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

```
ex1' :: (Arith_F ≺: f) ⇒ Fix_D f
ex1' = lit 3 `add` lit 4
```

The *prj* member function is a partial inverse of *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 *project* function is a variation of *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.

3.1.3 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 3.2. The signature *StuckValueF* represents a sentinel value to signal type errors during evaluation.

If f is a functor, we can fold over any value of type $Fix_D\ f$ as follows:

```
type Algebra f a = f a → a
fold_D :: Functor f ⇒ Algebra f a → Fix_D f → a
fold_D f (In_D x) = f (fmap (fold_D f) x)
```

```
data NatValueF   v = VInt   Int
data BoolValueF  v = VBool Bool
data StuckValueF v = VStuck


vint :: (NatValueF ≺: vf) ⇒ Int → Fix_D vf
vint i = inject (VInt i)
vbool :: (BoolValueF ≺: vf) ⇒ Bool → Fix_D vf
vbool b = inject (VBool b)
vstuck :: (StuckValueF ≺: vf) ⇒ Fix_D vf
vstuck = inject VStuck
```

Figure 3.2: Modular value datatype

An algebra specifies one step of recursion that turns a value of type f a into the desired result type a. The fold uniformly applies this operation to an entire term. All semantic functions over a modular datatype are written as folds of an algebra.

Using type classes, we can define and assemble algebras in a modular fashion. The class *FAlgebra* in Figure 3.3 carries an algebra for a functor f and carrier type a. It is additionally indexed over a parameter *name* to allow definitions of distinct functions with the same carrier. For instance, functions for calculating the size and the height of a term can both be defined using *Int* as the carrier.

We use the name *Eval* to refer to the evaluation algebra.

```
data Eval = Eval
```

The evaluation algebras are parameterized over an abstract super-functor *vf* for values. In case of $Arith_F$ we require that integral values are part of *vf* and for $Logic_F$ we require that boolean values are part of *vf*.

In the case of an *Add* in the evaluation algebra for arithmetic expressions we need to project the results of the recursive calls to test whether integral values were produced. Otherwise a type error occurrs and the *stuck* value is returned.

```
instance (NatValueF ≺: vf, StuckValueF ≺: vf) ⇒
         FAlgebra Eval Arith_F (Fix_D vf) where
```

```
class FAlgebra name f a where
  f_algebra :: name → Algebra f a

algebraPlus :: Algebra f a → Algebra g a →
                 Algebra (f ⊕ g) a
algebraPlus f g (Inl a) = f a
algebraPlus f g (Inr a) = g a

instance (FAlgebra name f a, FAlgebra name g a) ⇒
           FAlgebra name (f ⊕ g) a where
  f_algebra name = algebraPlus
                     (f_algebra name)
                     (f_algebra name)
```

Figure 3.3: Function algebra infrastructure

```
f_algebra Eval (Lit i)     = vint i
f_algebra Eval (Add a b) =
  case (project a, project b) of
    (Just (VInt a), Just (VInt b)) → vint (a + b)
    _                              → vstuck
```

Similarly, we have to test the result of the recursive call of the condition of an *If* term for boolean values.

```
instance (BoolValueF ≺: vf, StuckValueF ≺: vf) ⇒
           FAlgebra Eval Logic_F (Fix_D vf) where
  f_algebra Eval (BLit b)  = vbool b
  f_algebra Eval (If c t e) =
    case project c of
      Just (VBool b) → if b then t else e
```

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 3.3. Finally, we can define an evaluation function for terms given an *FAlgebra* instance for *Eval*.

$$
\begin{aligned}
&[\![\cdot]\!] :: (\mathit{Functor}\ f, \mathit{FAlgebra}\ \mathit{Eval}\ f\ (\mathit{Fix}_D\ \mathit{vf})) \Rightarrow \\
&\qquad \mathit{Fix}_D\ f \to \mathit{Fix}_D\ \mathit{vf} \\
&[\![\cdot]\!] = \mathit{fold}_D\ (f\_\mathit{algebra}\ \mathit{Eval})
\end{aligned}
$$

3.2 Reasoning à la Carte

In Section 3.1 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.

3.2.1 Modular Definitions in Coq

Unfortunately, we cannot directly translate the definitions of Section 3.1. Coq requires all inductive definitions to be *strictly-positive*. We define *strictly positive types* (SPT) by using the following generative grammar [1]:

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

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 $+$, $\times$, $\to$ and μ represent coproduct, cartesian product, exponentiation and least fixed point construction.

For $\mathit{Fix}_D\ f$ 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 $X \mapsto (X \to \mathit{Int}) \to \mathit{Int}$ into Fix_D yields the invalid datatype definition

$$\textbf{data}\ \mathit{NSP} = \mathit{NSP}\ ((\mathit{NSP} \to \mathit{Int}) \to \mathit{Int})$$

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 [**?**] 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 *SPF* type class in Figure 3.4 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 in_F and out_F as members that fold/unfold one layer of the fixpoint.

The fold operator from Section 3.1 also causes problems in Coq. *SPF* is a sub-class of *Functor* so we would like to define a generic fold operator similar to $fold_D$.

$$fold_F :: SPF\ f \Rightarrow Algebra\ f\ a \to Fix_F\ f \to a$$
$$fold_F\ alg = alg \circ fmap\ (fold_F\ alg) \circ out_F$$

Unfortunately, this definition is not structurally recursive and Coq is not able to determine its termination automatically. Hence, this definition is rejected. This is similar to the problem of Fix_F. For any concrete functor we can inline the definition of *fmap* to let $fold_F$ pass the termination check, but again we are working with an abstract functor f and an abstract functorial mapping *fmap*. Similarly, we resolve this by including a witness for the existence of a valid fold operator in the *SPF* class.

3.2.2 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 ind_A for arithmetic expression.

$$\begin{aligned} ind_A :: \forall p\ &:: (Arith \to Prop). \\ &\forall hl\ :: (\forall n. \qquad\qquad\quad p\ (in_F\ (Lit\ n))). \\ &\forall ha :: (\forall x\ y.p\ x \to p\ y \to p\ (in_F\ (Add\ x\ y))). \\ &\forall x.p\ x \end{aligned}$$

It takes a proposition p as parameter and inductive steps hl and ha for each case of the initial algebra. We say that hl and ha together form a *proof*

```
class Functor f ⇒ PFunctor f where
  type All :: ∀a.(a → Prop) → f a → Prop
type PAlgebra f a (alg :: Algebra f a) (p :: a → Prop) =
  PFunctor f ⇒ ∀(xs :: f a).All p xs → p (alg xs)
class PFunctor f ⇒ SPF (f :: ∗ → ∗) where
    -- Programming interface
  type Fix_F          :: ∗
  in_F                :: f (Fix_F f) → Fix_F f
  out_F               :: Fix_F f → f (Fix_F f)
  fold                :: Algebra f a → Fix_F f → a
    -- Reasoning interface
  in_out_inverse      :: ∀e.in_F (out_F e) = e
  out_in_inverse      :: ∀e.out_F (in_F e) = e
  fold_uniqueness     ::
    ∀a (alg :: Algebra f a) h.
      (∀e.h (in_F e) = alg (fmap h e)) →
      ∀x.h x = fold alg x
  fold_computation ::
    ∀a (alg :: Algebra f a) (x :: a),
      fold alg (in_F x) = alg (fmap (fold alg) x)
  ind                 ::
    ∀p.PAlgebra in_F p → ∀x.p x
```

Figure 3.4: Strictly-positive functor class

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* [**?**] for functors to capture the proofs of recursive positions. Informally, the all modality of a functor f and a predicate $p :: a \to Prop$ is a new type $All\ a\ p :: f\ a \to Prop$ that says that the predicate p holds for each $x :: a$ in an $f\ a$. The following type $Arith_{All}$ is an example of an all modality for arithmetic expressions.

$$\begin{array}{l}\textbf{data}\ Arith_{All}\ a\ p :: Arith_F\ a \to Prop\ \textbf{where}\\ \quad ALit\ \ :: \qquad\qquad\quad Arith_{All}\ a\ p\ (Lit\ n)\\ \quad AAdd :: p\ x \to p\ y \to Arith_{All}\ a\ p\ (Add\ x\ y)\end{array}$$

We introduce a new typeclass *PFunctor* that carries the associated all modality type and make *SPF* a subclass of it. Using the all modality definition we can write ind_A equivalently as

$$\begin{array}{l}ind_A\prime :: \forall p :: (Arith \to Prop).\\ \qquad\quad \forall h :: (\forall xs.Arith_{All}\ p\ xs \to p\ (in_F\ xs)).\\ \qquad\quad \forall x.p\ x\end{array}$$

The proof algebra is now a single parameter h. Note that h shows that p holds for an application of the initial algebra 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 3.4 more generally over arbitrary algebras. As a last member of *SPF* we introduce *ind* that is an induction principle for the fixpoint type Fix_F. It takes a proof algebra of a property p for the initial algebra and constructs a proof for every value of Fix_F.

3.2.3 Composing features

In Section 3.1 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 *PFunctor* class also has the nice property of being closed under coproducts.

$$\begin{array}{l}\textbf{instance}\ (PFunctor\ f, PFunctor\ g) \Rightarrow\\ \qquad\qquad PFunctor\ (f \oplus g)\ \textbf{where}\\ \quad \textbf{type}\ All\ a\ p\ xs = \textbf{case}\ xs\ \textbf{of}\\ \qquad\qquad\qquad\qquad Inl\ xs\ \to All\ a\ p\ xs\\ \qquad\qquad\qquad\qquad Inr\ xs \to All\ a\ p\ xs\end{array}$$

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

$$\begin{array}{l}\textbf{class}\ ProofAlgebra\ f\ a\ alg\ p\ \textbf{where}\\ \quad palgebra :: PAlgebra\ f\ a\ alg\ p\end{array}$$

```
instance (ProofAlgebra f a falg p,
          ProofAlgebra g a galg p) ⇒
  ProofAlgebra (f ⊕ a) a
    (algebraPlus falg galg) p where
  palgebra (Inl xs) axs = palgebra xs axs
  palgebra (Inr xs) axs = palgebra xs axs
```

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.

3.3 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 [**?**, **?**]. 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.

```
data Cont where
  (▷) :: (s :: ∗) → (p :: s → ∗) → Cont
shape (s ▷ p) = s
pos   (s ▷ p) = p
data Ext (c :: Cont) (a :: ∗) where
  Ext :: (s :: shape c) → (pos c s → a) → Ext c a
```

Figure 3.5: Container extension

3.3.1 Universe

The codes of the container universe are of the form $S \triangleright P$ where S denotes a type of shapes and $P :: S \to *$ denotes a family of position types indexed by S. The extension $Ext\ c$ of a container c in Figure 3.5 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 *gfmap* generically for any container.

```
gfmap :: (a → b) → Ext c a → Ext c b
gfmap f (Ext s pf) = Ext s (λp → f (pf p))
```

The functor $Arith_F$ 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 → ∗where
  Add_P1 :: Arith_P Add_S
  Add_P2 :: Arith_P Add_S
type Arith_C = Arith_S ▷ Arith_P
```

The shape of an $Arith_F$ 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_{P1} and Add_{P2}. *Lit* does not have any recursive positions.

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

```
from :: Arith_F a → Ext Arith_C a
from (Lit i) = Ext (Lit_S i) (λp → case p of )
```

$$
\begin{aligned}
&\mathit{from}\ (\mathit{Add}\ x\ y) = \mathit{Ext}\ \mathit{Add}_S\ \mathit{pf}\\
&\quad \textbf{where}\ \mathit{pf} :: \mathit{Arith}_P\ \mathit{Add}_S \to a\\
&\qquad\quad \mathit{pf}\ \mathit{Add}_{P1} = x\\
&\qquad\quad \mathit{pf}\ \mathit{Add}_{P2} = y\\
&\mathit{to} :: \mathit{Ext}\ \mathit{Arith}_C\ a \to \mathit{Arith}_F\ a\\
&\mathit{to}\ (\mathit{Ext}\ (\mathit{Lit}_S\ i)\ \mathit{pf}) = \mathit{Lit}\ i\\
&\mathit{to}\ (\mathit{Ext}\ \mathit{Add}_S\quad \mathit{pf}) = \mathit{Add}\ (\mathit{pf}\ \mathit{Add}_{P1})\ (\mathit{pf}\ \mathit{Add}_{P2})
\end{aligned}
$$

Literals do not have recursive positions and hence we cannot come up with a value. In Coq one needs to refute the position value $p :: \mathit{Arith}_P\ (\mathit{Lit}\ i)$ as its type is uninhabited. We use a case distinction without alternatives as an elimination.

3.3.2 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.

$$\textbf{data}\ W\ (c :: \mathit{Cont}) = \mathit{Sup}\ \{\mathit{unSup} :: \mathit{Ext}\ c\ (W\ c)\}$$

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.

$$
\begin{aligned}
&\mathit{gfold} :: \mathit{Algebra}\ (\mathit{Ext}\ c)\ a \to W\ c \to a\\
&\mathit{gfold}\ \mathit{alg}\ (\mathit{Sup}\ (\mathit{Ext}\ s\ \mathit{pf})) =\\
&\quad \mathit{alg}\ (\mathit{Ext}\ s\ (\lambda p \to \mathit{gfold}\ \mathit{alg}\ (\mathit{pf}\ p)))
\end{aligned}
$$

Note that this definition is essentially the same as the definition of fold_D from Section 3.1. Because of the generic implementation of *gfmap* we can inline it to expose the structural recursion. Coq accepts this definition, since the recursive call $\mathit{gfold}\ \mathit{alg}\ (\mathit{pf}\ p)$ is performed on the structurally smaller argument $\mathit{pf}\ p$.

3.3.3 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.

$$
\begin{aligned}
&\textbf{type}\ S_+ = \mathit{Either}\ S_1\ S_2\\
&\textbf{type}\ P_+\ (\mathit{Left}\ s) \quad = P_1\ s\\
&\textbf{type}\ P_+\ (\mathit{Right}\ s) = P_2\ s
\end{aligned}
$$

$$
\begin{array}{l}
\textbf{class}\ \mathit{Container}\ (f :: * \to *)\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{cont} & :: \mathit{Cont} \\
\mathit{from} & :: f\ a \to \mathit{Ext}\ c\ a \\
\mathit{to} & :: \mathit{Ext}\ c\ a \to f\ a \\
\mathit{fromTo} & :: \forall x.\mathit{from}\ (\mathit{to}\ x) \equiv x \\
\mathit{toFrom} & :: \forall x.\mathit{to}\ (\mathit{from}\ x) \equiv x
\end{array}
\end{array}
$$

Figure 3.6: Container functor class

The injection functions on the extensions are given by

$$
\begin{array}{l}
\mathit{inl} :: \mathit{Ext}\ (S_1 \triangleright P_1) \to \mathit{Ext}\ (S_+ \triangleright P_+) \\
\mathit{inl}\ (\mathit{Ext}\ s\ pf) = \mathit{Ext}\ (\mathit{Left}\ s)\ pf \\
\mathit{inr} :: \mathit{Ext}\ (S_2 \triangleright P_2) \to \mathit{Ext}\ (S_+ \triangleright P_+) \\
\mathit{inr}\ (\mathit{Ext}\ s\ pf) = \mathit{Ext}\ (\mathit{Right}\ s)\ pf
\end{array}
$$

3.3.4 Induction

To define an induction principle for container types we proceed in the same way as in Section 3.2.2 by defining proof algebras using an *all modality* [**?**]. The all modality on containers is given generically by a Π-type that asserts that q holds at all positions.

$$
\begin{array}{l}
\mathit{GAll} :: (q :: a \to \mathit{Prop}) \to \mathit{Ext}\ c\ a \to \mathit{Prop} \\
\mathit{GAll}\ q\ (\mathit{Ext}\ s\ pf) = \forall (p :: \mathit{pos}\ c\ s).q\ (pf\ p)
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{gind} :: \forall (c :: \mathit{Cont}) \to \\
\qquad \forall (q :: W\ c \to \mathit{Prop}) \to \\
\qquad \forall (\mathit{palg} :: \forall xs.\mathit{GAll}\ q\ xs \to q\ (\mathit{Sup}\ xs)) \to \\
\qquad \forall x.q\ x \\
\mathit{gind}\ c\ q\ \mathit{palg}\ (\mathit{Sup}\ (\mathit{Ext}\ s\ pf)) = \\
\quad \mathit{palg}\ (\lambda p \to \mathit{gind}\ c\ q\ \mathit{palg}\ (pf\ p))
\end{array}
$$

3.3.5 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 $* \to *$ as long as it is isomorphic to a container functor. The type class *Container* in Figure 3.6 witnesses this isomorphism. The class contains the functions *from* and *to* that perform the conversion between a conventional functor and a container functor and proofs that these conversions are inverses.

Via the isomorphisms *from* and *to* we can import all the generic functions to concrete functors and give instances for *Functor*, *PFunctor* and *SPF*.

```
instance Container f ⇒ Functor f where
  fmap f   = to ∘ gfmap f ∘ from
instance Container f ⇒ PFunctor f where
  All Q    = GAll Q ∘ from
instance Container f ⇒ SPF f where
  Fix_F    = W S P
  in_F     = sup ∘ from
  out_F    = to ∘ unSup
  fold alg = gfold (alg ∘ to)
  ...
```

The important difference to the *SPF* class is that we can generically build the instance for the coproduct of two *Container* functors by using the coproduct of their containers.

```
instance (Container f, Container g) ⇒
  Container (f ⊕ g)
```

3.3.6 Extensible logical relations

Many properties are expressed as logical relations over datatypes. These relations are represented by inductive families where a constructor 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. For instance, a well-typing relation of values $WTValue :: (Value, Type) \to Prop$ must be extended with new rules when new cases are added to *Value*.

Extensibility of inductive families is obtained in the same way as for inductive datatypes by modularly building inductive families as fixpoints of functors

```
class IFunctor i (f :: (i → Prop) → i → Prop) where
  ifmap :: ∀(a :: i → Prop) (b :: i → Prop) (j :: i).
             (∀j.a j → b j) → f a j → f b j
class IFunctor i f ⇒
      ISPF i (f :: (i → Prop) → i → Prop) where
  type IFix :: i → Prop
  in_IF     :: ∀(j :: i).f (IFix f i) → IFix f i
  out_IF    :: ∀(j :: i).IFix f i → f (IFix f i)
  ifold     :: IAlgebra i f a → ∀j.IFix f j → a j
```

Figure 3.7: Indexed strictly-positive functor class

between inductive families. The following indexed functor $WTNat_F$ covers the rule that a natural number value has a natural number type.

```
data WTNat_F (wfv :: (Fix_F vf, Fix_F tf) → Prop) ::
  (Value, Type) → Prop where
  WTNat :: (NatValueF ≺: vf, NatTypeF ≺: tf) ⇒
    WTNat wfv (vi n, tnat)
```

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

```
type IAlgebra i (f :: (i → Prop) → i → Prop) a =
  ∀(j :: i).f a j → a j
type IFix_M i (f :: (i → Prop) → i → Prop) j =
  ∀a.IAlgebra i f a → 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 3.7 defines the type class *ISPF* that collects the necessary functions for modularly building logical relations.

Alternatively we can use a universe of indexed containers [2]. An i-indexed container is essentially a container together with an assignment of indices for each shape and each position of that shape.

```
data ICont i where
  (_ ▷ _ ▷ _) :: (s :: i → ∗) →
                 (p :: ∀j.s j → ∗) →
                 (r :: ∀j s.p j s → i) → ICont i
ishape  (s ▷ p ▷ r) = s
ipos    (s ▷ p ▷ r) = p
irec    (s ▷ p ▷ r) = r
data IExt (c :: ICont i)
          (a :: i → Prop) (j :: i) :: Prop where
  IExt :: (s  :: ishape c j) →
          (pf :: ∀(p :: ipos c j s).a (irec c j s p)) →
          IExt c a j
data IW (c :: ICont i) (j :: i) :: Prop where
  ISup :: IExt c IW j → IW c j
```

Figure 3.8: i-indexed containers

More formally, an i-indexed container $S \triangleright P \triangleright R$ is given by a family of shapes $S :: i \to *$ and family of position types $P :: (j :: i) \to S\ j \to *$ and an assignment $R :: (j :: i) \to (s :: S\ j) \to P\ j\ s \to i$ of indices for positions. Figure 3.8 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 3.3.2. Indexed containers are also closed under coproducts and indexed algebras can be modularly composed using type classes.

3.4 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 implementations as well. In this section we look at a generic implementation of equality testing and proofs about its correctness. These functions are used for example in the MTC framework in the implementation of a modular type-checker that tests if both branches of an **if** expression have the same type and that the function

```
class Eq a where
  eq      :: a → a → Bool
  eqTrue  :: ∀x y. eq x y = True  → xs = ys
  eqFalse :: ∀x y. eq x y = False → xs ≠ ys
```

Figure 3.9: Equality type class

and argument type of a function application are compatible. Furthermore for reasoning about functions that use equality testing we need proofs about its correctness. We thus include the equality function and the properties in an equality type class that is shown in Figure 3.9.

When choosing an approach to generic programming there is a trade-off between the expressivity of the approach, i.e. the collection of types it covers, and the functionality that can be implemented generically using this approach. The container universe is a very expressive universe for which we have generically implemented folds and hence is well-suited as a solution for modularly defining datatypes and functions. However, it is too expressive for implementing equality generically as it also includes function types.

So instead we restrict ourselves to the universe of polynomial functors to implement equality generically.

3.4.1 Universe

The codes *Poly* and interpretation Ext_P of the polynomial functor universe are shown in Figure 3.10. A polynomial functor is either the constant unit functor U, the identity functor I, a coproduct $C\ p_1\ p_2$ of two functors, or the cartesian product $P\ p_1\ p_2$ 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)
```

```
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 a → Ext_P (C c d) a
  ER :: Ext_P d a → Ext_P (C c d) a
  EP :: Ext_P c a → Ext_P d 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
  ptoFromInverse :: ∀a.pto (pfrom a) = a
  pfromToInverse :: ∀a.pfrom (pto a) = a
```

Figure 3.10: Polynomial functors

```
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 3.10. An instance for *FunType* is the following, with proofs omitted:

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

3.4.2 Universe embedding

To write modular functions for polynomial functors we proceed in the same way as in Section 3.3 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 recursing over the code.

$$
\begin{aligned}
&poly_S :: Poly \to * \\
&poly_S\ U &&= () \\
&poly_S\ I &&= () \\
&poly_S\ (C\ c\ d) &&= poly_S\ c + poly_S\ d \\
&poly_S\ (P\ c\ d) &&= (poly_S\ c, poly_S\ d)
\end{aligned}
$$

The constant unit functor and the identity functor have only one shape which is represented by a unit type. As in section 3.3.3 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 recursing over the code.

$$
\begin{aligned}
&poly_P :: (c :: Poly) \to poly_S\ c \to * \\
&poly_P\ U\ ()\ &&= Empty \\
&poly_P\ I\ ()\ &&= () \\
&poly_P\ (C\ c\ d)\ (Left\ s)\ &&= poly_P\ c\ s \\
&poly_P\ (C\ c\ d)\ (Right\ s)\ &&= poly_P\ d\ s \\
&poly_P\ (P\ c\ d)\ (s_1, s_2)\ &&= \\
&\quad Either\ (poly_P\ c\ s_1)\ (poly_P\ d\ s_2)
\end{aligned}
$$

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 *ptoUnary* converts the polynomial interpretation to the container intepretation.

$$
\begin{aligned}
&ptoUnary :: (c :: Poly) \to \\
&\qquad Ext_P\ c\ a \to Ext\ (poly_S\ c \triangleright poly_P\ c)\ a \\
&ptoUnary\ U\ EU\ &&= Ext\ ()\ (\lambda p \to \mathbf{case}\ p\ \mathbf{of}\ ) \\
&ptoUnary\ I\ (EI\ a)\ &&= Ext\ ()\ (\lambda() \to a) \\
&ptoUnary\ (C\ c\ d)\ (EL\ x)\ &&= Ext\ (Left\ s)\ pf \\
&\quad \mathbf{where}\ Ext\ s\ pf = ptoUnary\ c\ x
\end{aligned}
$$

```
ptoUnary (C c d) (ER y)    = Ext (Right s) pf
  where Ext s pf = ptoUnary c y
ptoUnary (P c d) (EP x y) = Ext (s1, s2)
                                (λp → case p of
                                  Left p  → pf1 p
                                  Right p → pf2 p)
  where Ext s1 pf1 = ptoUnary c x
        Ext s2 pf2 = ptoUnary d y
```

Similarly we define the function *pfromUnary* that performs the conversion in the opposite direction. We omit the implementation.

```
pfromUnary ::
  (c :: Poly) → Ext (polyS c ▷ polyP c) a → ExtP c a
```

To transport properties across these conversion functions we need to prove that they are inverses. These proofs proceed by inducting over the code; we omit them here.

As the last step we derive an instance of *Container* from an instance of *Polynomial*. This way all the generic functionality of containers is also available for polynomial functors.

```
instance Polynomial f ⇒ Container f where
  cont    = polyS pcode ▷ polyP pcode
  from    = ptoUnary pcode ∘ pfrom
  to      = pto ∘ pfromUnary pcode
  fromTo = ...
  toFrom = ...
```

3.4.3 Generic equality

Performing the conversions between polynomial functors and containers in the definition of recursive functions makes it difficult to convince the termination checker to accept these definitions. So instead of using the generic fixed point provided by the container universe we define a generic fixed point on the polynomial functor universe directly.

```
data FixP (c :: Poly) = FixP (ExtP c (FixP c))
```

We define the generic equality function *geq* mutually recursively with *go* that recurses over the codes and forms an equality function for a partially constructed fixed point.

$$
\begin{aligned}
&geq :: (c :: Poly) \to Fix_P\ c \to Fix_P\ c \to Bool\\
&geq\ c\ (Fix_P\ x)\ (Fix_P\ y) = go\ c\ x\ y\\
&\quad \textbf{where}\\
&\quad\quad go :: (d :: Poly) \to\\
&\quad\quad\quad\quad Ext_P\ d\ (Fix_P\ c) \to Ext_P\ d\ (Fix_P\ c) \to Bool\\
&\quad\quad go\ U\ \ \ \ \ \ EU\ \ \ \ \ \ \ EU\ \ \ \ \ \ \ = True\\
&\quad\quad go\ I\ \ \ \ \ \ (EI\ x)\ \ \ \ (EI\ y)\ \ \ \ = geq\ x\ y\\
&\quad\quad go\ (C\ c\ d)\ (EL\ x)\ \ \ \ (EL\ y)\ \ \ \ = go\ c\ x\ y\\
&\quad\quad go\ (C\ c\ d)\ (EL\ x)\ \ \ \ (ER\ y)\ \ \ \ = False\\
&\quad\quad go\ (C\ c\ d)\ (ER\ x)\ \ \ \ (EL\ y)\ \ \ \ = False\\
&\quad\quad go\ (C\ c\ d)\ (ER\ x)\ \ \ \ (ER\ y)\ \ \ \ = go\ d\ x\ y\\
&\quad\quad go\ (P\ c\ d)\ (EP\ x\ x')\ (EP\ y\ y') =\\
&\quad\quad\quad go\ c\ x\ y \wedge go\ d\ x'\ y'
\end{aligned}
$$

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.

Of course $Fix_P\ c$ and $Fix_F\ (poly_S\ c \triangleright poly_P\ c)$ are isomorphic and we can transport functions and their properties across this isomorphism to get a generic equality function on the fixed point defined by containers for a conventional polynomial functor which can be used to instantiate the *Eq* type class in Figure 3.9.

$$\textbf{instance}\ Polynomial\ f \Rightarrow Eq\ (Fix_F\ f)$$

3.5 Case study

As a demonstration of the advantages of our approach over MTC's Church-encoding based approach, we have ported the case study from [**?**].

The study consists of five reusable language features with soundness and continuity[2] proofs in addition to typing and evaluation functions. Figure 3.11 presents the syntax of the expressions, values, and types provided by the features; each line is annotated with the feature that provides that set of definitions.

In this section we discuss the benefits and trade-offs we have experienced while porting the case study to our approach.

[2] of step-bounded evaluation functions

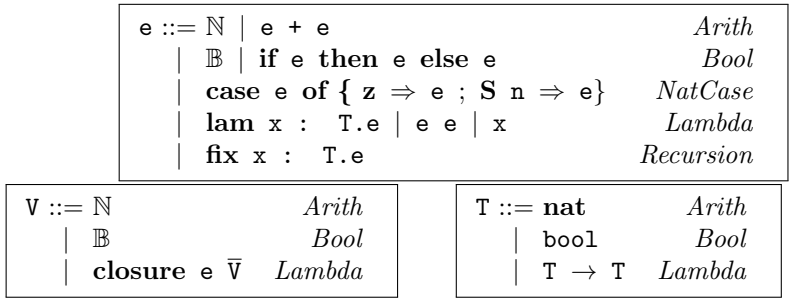

Figure 3.11: mini-ML expressions, values, and types

Code size 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.

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, instantiating the MTC interface amounts to roughly 40 LoC per feature while our approach requires about 100 LoC per feature for the instances.

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.

Impredicative sets The higher-rank type in the definition of Fix_M

$$Fix_M\ (f :: Set \to Set) = \forall (a :: Set).Algebra\ f\ a \to a$$

causes $Fix_M\ f$ to be in a higher universe level than the domain of f. Hence to use $Fix_M\ f$ 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 ind_A in Section 3.2.2. MTC uses a *poor-man's induction principle* ind_A^2 instead.

$$
\begin{array}{l}
ind_A^2 :: \\
\quad \forall(p \ :: (Arith \to Prop)). \\
\quad \forall(hl :: (\forall n.p\ (InMTC\ (Lit\ n)))). \\
\quad \forall(ha :: (\forall x\ y.p\ x \to p\ y \to p\ (InMTC\ (Add\ x\ y)))). \\
\quad Algebra\ Arith_F\ (\exists a.p\ a)
\end{array}
$$

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-classes 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.

$$
\begin{array}{l}
foldMTC :: Algebra\ f\ a \to Fix_M\ f \to a \\
foldMTC\ alg\ fa = fa\ alg \\
\textbf{type}\ UniversalProperty\ (f :: * \to *)\ (e :: Fix_M\ f) \\
\quad = \forall a\ (alg :: Algebra\ f\ a)\ (h :: Fix_M\ f \to a). \\
\qquad (\forall e.h\ (inMTC\ e) = alg\ h\ e) \to \\
\qquad\quad h\ e = foldMTC\ alg\ e
\end{array}
$$

   In an initial algebra representation of an inductive datatype, we have a single implementation of a fold operator that can be proven correct. In MTC's approach based on Church-encodings however, each term consists of a separate fold implementation that must satisfy the universal property.

Hence, in order to enable reasoning MTC must provide a proof of the universal property of folds for every value of a modular datatype that is used in a proof. This is mostly done by packaging a term and the proof of the universal property of its fold in a dependent sum type.

$$\textbf{type } \mathit{FixUP}\ f = \exists(x :: \mathit{Fix}_M\ f).\, \mathit{UniversalProperty}\ f\ x$$

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 $\mathit{WTValue}\ (v, t)$ 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 $\mathit{WTValue}\ (v, t)$ and $\mathit{proj1}\ t = \mathit{proj1}\ t'$ does not directly imply $\mathit{WTValue}\ (v, t')$ 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 functor f the folding $\mathit{inMTC} :: f\ (\mathit{Fix}_M\ f) \to \mathit{Fix}_M\ f$ and unfolding $\mathit{outMTC} :: \mathit{Fix}_M\ f \to f\ (\mathit{Fix}_M\ f)$ are inverse operations, namely, that all appearing $\mathit{Fix}_M\ f$ values need to have the universal property of folds. This side-condition raises the question if $\mathit{Fix}_M\ f$ 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 *SPF* type class from Figure 3.4 requires that **in** and *out* are inverse operations without any side conditions on the values and containers give rise to proper *SPF* instances.

3.6 Related and Future Work

DGP in proof-assistants Datatype-generic programming started out as a form of language extension such as PolyP [**?**] or Generic Haskell [**?**]. 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 [**?**, **?**, **?**, **?**, **?**, **?**, **?**]. 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 [**?**, **?**, **?**, **?**, 2]. 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 [**?**]. 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 [**?**] 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 [**?**]. Magalhães and Löh [**?**] 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 [**?**, 4]. 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 [**?**] and Keuchel and Jeuring [**?**] define universes for datatypes with binding in Agda. Lee et al. [3] develop a framework for first-order representations of variable binding in Coq that is based on the universe of regular tree types [**?**] 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 [3] 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.

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 encodes fixed points and leads to stronger induction principles by exploiting DGP approaches that capture only strictly-positive functors.

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

Chapter 4

Modular monadic effects

Chapter 5

Conclusions

Summary

Nederlandstalige samenvatting

Bibliography

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In *FOSSACS*, volume 2620 of *LNCS*, pages 23–38. Springer, 2003. ↑14, ↑18

[2] T. Altenkirch and P. Morris. Indexed containers. In *LICS '09*, pages 277–285, 2009. ↑23, ↑33

[3] Gyesik Lee, Bruno C.D.S. Oliveira, Sungkeun Cho, and Kwangkeun Yi. GMeta: A generic formal metatheory framework for first-order representations. In *ESOP*. Springer, 2012. ↑34

[4] Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. Binders unbound. In *ICFP '11*. ACM, 2011. ↑34

List of Figures

List of Tables