Meta-Theory à la Carte

Benjamin Delaware
University of Texas at Austin
bendy@cs.utexas.edu

Bruno C. d. S. Oliveira
National University of Singapore
oliveira@comp.nus.edu.sg

Tom Schrijvers
Universiteit Gent
tom.schrijvers@ugent.be

Abstract

Formalizing meta-theory, or proofs about programming languages, in a proof assistant has many well-known benefits. However, the considerable effort involved in mechanizing proofs has prevented it from becoming standard practice. This cost can be amortized by reusing as much of an existing formalization as possible when building a new language or extending an existing one. Unfortunately reuse of components is typically ad-hoc, with the language designer cutting and pasting existing definitions and proofs, and expending considerable effort to patch up the results.

This paper presents a more structured approach to the reuse of formalizations of programming language semantics through the composition of modular definitions and proofs. The key contribution is the development of an approach to induction for extensible Church encodings which uses a novel reinterpretation of the universal property of folds. These encodings provide the foundation for a framework, formalized in Coq, which uses type classes to automate the composition of proofs from modular components.

Several interesting language features, including binders and general recursion, illustrate the capabilities of our framework. We reuse these features to build fully mechanized definitions and proofs for a number of languages, including a version of mini-ML. Bounded induction enables proofs of properties for non-inductive semantic functions, and mediating type classes enable proof adaptation for more feature-rich languages.

1. Introduction

With their POPLMARK challenge, Aydemir et al. [14] identified *representation of binders*, *complex inductions*, *experimentation*, and *reuse of components* as key challenges in mechanizing programming language meta-theory. While progress has been made, for example on the representation of binders, it is still difficult to reuse components, including language definitions and proofs.

The current approach to reuse still involves copying an existing formalization and adapting it manually to incorporate new features. An extreme case of this copy-&-adapt approach can be found in Leroy's 3 person-year verified compiler project [22]: it consists of 8 intermediate languages in addition to the source and target languages, many of which are minor variations of each other. Due to the crosscutting impact of new features, the adaptation of existing features is still unnecessarily labor-intensive. Moreover, from a software/formalization management perspective a proliferation of copies is obviously a nightmare.

There are two main challenges in providing reuse:

1. **Extensibility:** Conventional proofs and definitions are *closed* to extension: the proofs and definitions of a language cannot simply be imported and extended with new constructs. This is a manifestation of the well-known *Expression Problem* (EP) [42].
2. **Modular reasoning:** Reasoning with modular definitions requires reasoning about partial definitions and composing partial proofs to obtain a complete proof. However, most reasoning techniques (such as induction) assume complete definitions.

The lack of reuse in formalizations is somewhat surprising, because proof assistants such as Coq and Agda have powerful modularity constructs including *modules* [26], *type classes* [17, 39, 43] and expressive forms of *dependent types* [10, 34]. It is reasonable to wonder whether these language constructs can achieve better reuse. After all, there has been a lot of progress in addressing extensibility [13, 30, 32, 40] issues in general-purpose languages using advanced type system features – although not a lot of attention has been paid to modular reasoning.

This paper presents MTC, a framework for defining modular meta-theory which shows that proof assistants can effectively support component reuse. MTC deals with both extensibility and modular reasoning and is implemented as a Coq library. MTC enables writing modular definitions and proofs. To formalize a new language or language extension, the bulk of the language can be assembled from previously written components such as binders, with related operations and proofs. As such users can focus on developing the truly new and interesting features.

The solution to extensibility in MTC was partly inspired by the popular "Data types à la Carte" (DTC) [40] technique. However adapting DTC to Coq, which imposes significant restrictions on recursive definitions to ensure termination, proved challenging. DTC fundamentally relies on a type-level fixpoint definition for building modular data types, but this type-level fixpoint cannot be encoded in Coq. MTC solves this problem by using *Church encodings* of data types [34, 35] instead. These Church encodings allow us to define modular data types in the DTC-style in the restricted Coq setting. Another difference between DTC and MTC is the use of Mendler-style folds and algebras instead of conventional folds to express modular definitions. The advantage of Mendler-style folds [41] and algebras is that they offer explicit control over the evaluation order, which is important when modeling semantics of programming languages. MTC employs similar techniques to solve extensibility problems in proofs and *inductive relations*.

MTC's solution to modular reasoning uses a novel reinterpretation of the universal property of folds. Proof methods such as structural induction cannot be used since they require closed or complete definitions. However, because MTC relies on folds, the proof methods used in the *initial algebra semantics of data types* [16, 27] offered an initial handle on this problem. With some care and adaptations, *universal properties* and other derived principles work quite well with modular Church encodings. Not only do universal properties provide a modular way to reason about proofs, but they also overcome some theoretical issues related to Church encodings in the Calculus of (Inductive) Constructions [34, 35].

Ubiquitous higher-order features such as binders and general recursion can also be implemented in MTC. Binders are modeled with a *parametric HOAS* [7] representation (a first-order representation would be possible too). Because such higher-order features

require general recursion, they cannot be defined inductively using folds. To support these non-inductive features MTC uses a variation of mixins [9]. Mixins are closely related to Mendler-style folds, but they allow uses of general recursion, and can be modeled on top of Mendler-style Church encodings using a bounded fixpoint combinator.

To illustrate MTC, we present a case study modularizing several orthogonal features of a variant of mini-ML [8]. The case study illustrates how various features and partial *type soundness* proofs can be modularly developed and verified and later composed to assemble complete languages and proofs.

1.1 Contributions

The main contribution of our work is a novel approach to mechanizing meta-theory, which allows *modular* development of both semantic definitions and their proofs. MTC is a framework of reusable semantic components that builds on this approach and allows defining modular mechanized meta-theory in Coq.

More technically this paper makes the following contributions:

- **Extensibility Techniques for Mechanization:** The paper provides a solution to the EP and an approach to extensible mechanization of meta-theory in the restricted type-theoretic setting of Coq. This solution offers precise control over the evaluation order by means of Mendler folds and algebras. Mixins are used to capture ubiquitous higher-order features, like PHOAS binders and general recursion.
- **Non-Axiomatic Reasoning for Church Encodings:** The paper reinterprets the universal property of folds to recover induction over Mendler-style Church encodings. This allows us to avoid the axioms used in earlier approaches and preserves Coq's strong normalization.
- **Modular Reasoning:** The paper presents modular reasoning techniques for modular components. It lifts the recovered induction principle from individual inductive features to compositions, while induction over a bounded step count enables modular reasoning about non-inductive higher-order features modeled with mixins.

MTC is implemented in the Coq proof assistant and the code is available at `http://www.cs.utexas.edu/~bendy/MTC`. Our implementation minimizes the programmer's burden for adding new features by automating the boilerplate with type classes and default tactics. Moreover, the framework already provides modular components for mini-ML as a starting point for new language formalizations. We also provide a complimentary Haskell implementation of the computational subset of code used in this paper.

1.2 Code and Notational Conventions

While all the code underlying this paper has been developed in Coq, we have adopted a terser syntax for the many code fragments shown in this paper. For the computational parts, this syntax exactly coincides with Haskell syntax, while it is an extrapolation of Haskell syntax style for proposition- and proof-related concepts. The Coq code requires the impredicative-set option.

2. Extensible Semantics in MTC

This section shows MTC's approach to extensible and modular semantic components in the restrictive setting of Coq. The approach is partly inspired by DTC, which is a solution to the Expression Problem in Haskell that provides composition mechanisms for modular definitions. MTC differs from DTC in two important ways. Firstly, Church encodings are used to avoid the termination issues of DTC's generally recursive definitions. Secondly, Mendler-style folds are used instead of conventional folds to provide explicit control over evaluation order.

2.1 Data Types à la Carte

This subsection reviews the core ideas of DTC. DTC represents the shape of a particular data type as a *functor* f. That functor uses its type parameter a for inductive occurrences of the data type, leaving the data type definition open. $Arith_F$ is an example functor for a simple language of arithmetic expressions with literals and addition.

$$\mathbf{data}\ Arith_F\ a = Lit\ Nat \mid Add\ a\ a$$

The explicitly recursive definition $Fix\ f$ closes the open recursion of a functor f.

$$\mathbf{data}\ Fix_{DTC}\ f = In\ (f\ (Fix_{DTC}\ f))$$

Applying Fix_{DTC} to $Arith_F$ builds the data type for arithmetic expressions.

$$\mathbf{type}\ Arith = Fix_{DTC}\ Arith_F$$

Functions over $Fix_{DTC}\ f$ are expressed as folds of F-Algebras.

$$\mathbf{type}\ Algebra\ f\ a = f\ a \to a$$
$$fold_{DTC} :: Functor\ f \Rightarrow Algebra\ f\ a \to Fix_{DTC}\ f \to a$$
$$fold_{DTC}\ alg\ (In\ fa) = alg\ (fmap\ (fold_{DTC}\ alg)\ fa)$$

For example, the evaluation algebra of $Arith_F$ is defined as:

$$\mathbf{data}\ Value = I\ Int \mid B\ Bool$$
$$eval_{Arith} :: Algebra\ Arith_F\ Value$$
$$eval_{Arith}\ (Lit\ n) = I\ n$$
$$eval_{Arith}\ (Add\ (I\ v_1)\ (I\ v_2)) = I\ (v_1 + v_2)$$

Note that the recursive occurrences in $eval_{Arith}$ are of the same type as the result type $Value$.[1] In essence, folds process the recursive occurrences, so that algebras only need to specify how to combine the values (for example v_1 and v_2) resulting from evaluating the subterms. Finally, the overall evaluation function is:

$$[\![\cdot]\!] :: Fix_{DTC}\ A \to Value$$
$$[\![\cdot]\!] = fold_{DTC}\ eval_{Arith}$$

$$> [\![(Add\ (Lit\ 1)\ (Lit\ 2))]\!]$$
$$3$$

Unfortunately, DTC's double use of general recursion is not allowed by Coq. Coq does not accept the type-level fixpoint combinator $Fix_{DTC}\ f$ because it is not strictly positive. The $fold_{DTC}$ function is similarly problematic for Coq's termination checker, because the recursive call does not conform to Coq's structural restrictions.

2.2 Recursion-Free Church Encodings

MTC encodes data types and folds with Church encodings [5, 35], which are recursion-free. Church encodings represent (least) fixpoints and folds as follows:

$$\mathbf{type}\ Fix\ f = \forall a.Algebra\ f\ a \to a$$
$$fold :: Algebra\ f\ a \to Fix\ f \to a$$
$$fold\ alg\ fa = fa\ alg$$

Both definitions are non-recursive and can be encoded in Coq (although we need to enable impredicativity for certain definitions). Because Church encodings represent data types as folds, the definition of $fold$ is trivial: it applies the data type fold $Fix\ f$ to the algebra.

[1] Boolean values are not needed yet, but they will be useful later in this section.

For example, these are the Church encodings of $Arith_F$'s literals and addition:

$$lit :: Nat \to Fix\ Arith_F$$
$$lit\ n = \lambda alg \to alg\ (Lit\ n)$$
$$add :: Fix\ Arith_F \to Fix\ Arith_F \to Fix\ Arith_F$$
$$add\ e_1\ e_2 = \lambda alg \to alg\ (Add\ (fold\ alg\ e_1)\ (fold\ alg\ e_2))$$

The evaluation algebra and evaluation function are defined as in DTC, and expressions are evaluated in much the same way.

2.3 Lack of Control over Evaluation

Folds are structurally recursive and, as such, capture *compositionality* of definitions, a desirable property of semantics. A disadvantage of the standard fold encoding is that it does not provide the implementer of the algebra with explicit control of evaluation. The fold encoding reduces all subterms; the only freedom in the algebra is whether or not to use the result.

Example: Modeling* if *expressions As a simple example that illustrates the issue of lack of control over evaluation consider modeling **if** expressions and their corresponding semantics. The big-step semantics of **if** expressions is:

$$\frac{[\![e_1]\!] \rightsquigarrow \mathbf{true} \qquad [\![e_\mathbf{2}]\!] \rightsquigarrow v_\mathbf{2}}{[\![\ \mathbf{if}\ e_1\ e_2\ e_3]\!] \rightsquigarrow v_2} \qquad \frac{[\![e_1]\!] \rightsquigarrow \mathbf{false} \qquad [\![e_\mathbf{3}]\!] \rightsquigarrow v_\mathbf{3}}{[\![\ \mathbf{if}\ e_1\ e_2\ e_3]\!] \rightsquigarrow v_3}$$

Using our framework of Church encodings, we could create a modular feature for boolean expressions such as **if** expressions and boolean literals as follows:

$$\mathbf{data}\ Logic_F\ a = If\ a\ a\ a \quad \text{-- functor}$$
$$\qquad\qquad\qquad\qquad\ \mid BLit\ Bool$$
$$eval_{Logic} :: Algebra\ Logic_F\ Value$$
$$eval_{Logic}\ (If\ v_1\ v_2\ v_3) = \mathbf{if}\ (v_1 \equiv B\ True)\ \mathbf{then}\ v_2\ \mathbf{else}\ v_3$$
$$eval_{Logic}\ (BLit\ b) \qquad = B\ b$$

However, an important difference with the big-step semantics is that $eval_{Logic}$ cannot control where evaluation happens. All it has in hand are the values v_1, v_2 and v_3 that result from evaluation. While this difference is not important for simple features like arithmetic expressions, it does matter for **if** expressions.

Semantics guides the development of implementations. As such, we believe that it is important that a semantic specification does not rely on a particular evaluation strategy (such as laziness). This definition of $eval_{Logic}$ might be reasonable in a lazy meta-language like Haskell (which is the language used by DTC), but it is misleading when used as a basis for an implementation in a strict language like ML. In a strict language $eval_{Logic}$ is clearly not a desirable definition because it evaluates both branches of the **if** expression. Aside from the obvious performance drawbacks, this is the wrong thing to do if the object language features, for example, non-termination.

Using standard folds in a lazy language *often* recovers the desired semantics by delaying a computation until it is needed. However this relies on the particular evaluation order of the meta-language, which the can be misleading for someone trying to implement the semantics. Furthermore, this approach can be quite brittle: in more complex object languages using folds and laziness can lead to subtle semantic issues [3].

2.4 Mendler-style Church Encodings

To express semantics in a way that allows explicit control over evaluation and does not rely on the evaluation semantics of the meta-language, MTC adapts Church encodings to use Mendler-style algebras and folds [41] which make recursive calls explicit.

$$\mathbf{type}\ Algebra_M\ f\ a = \forall r.(r \to a) \to f\ r \to a$$

A Mendler-style algebra differs from a traditional F-algebra in that it takes an additional argument $(r \to a)$ which corresponds to recursive calls. To ensure that recursive calls can only be applied structurally, the types of the arguments that appear at recursive positions have a polymorphic type r. The use of this polymorphic type r prevents case analysis, or any other type of inspection, on those arguments. Using $Algebra_M\ f\ a$, Mendler-style folds and Mendler-style Church encodings are defined as follows:

$$\mathbf{type}\ Fix_M\ f = \forall a.Algebra_M\ f\ a \to a$$
$$fold_M :: Algebra_M\ f\ a \to Fix_M\ f \to a$$
$$fold_M\ alg\ fa = fa\ alg$$

Mendler-style folds allow algebras to state their recursive calls explicitly. For example, the definition of the evaluation of **if** expressions in terms of a Mendler-style algebra is:

$$eval_{Logic} :: Algebra_M\ Logic_F\ Value$$
$$eval_{Logic}\ [\![\cdot]\!]\ (BLit\ b) \qquad = B\ b$$
$$eval_{Logic}\ [\![\cdot]\!]\ (If\ e_1\ e_2\ e_3) = \mathbf{if}\ ([\![e_1]\!] \equiv B\ True)\ \mathbf{then}\ [\![e_2]\!]\ \mathbf{else}\ [\![e_3]\!]$$

Note that this definition allows explicit control over the evaluation order just like the big-step semantics definition. Furthermore, like the fold-definition, $eval_{Logic}$ enforces compositionality because all the algebra can do to e_1, e_2 or e_3 is to apply the recursive call $[\![\cdot]\!]$.

2.5 A Compositional Framework for Mendler-style Algebras

DTC provides a convenient framework for composing conventional fold algebras. MTC provides a similar framework, but for Mendler-style algebras instead of F-algebras. In order to write modular proofs about semantics, MTC needs to regulate its definitions with a number of laws.

Modular Functors Individual features can be modularly defined using functors, like $Arith_F$ and $Logic_F$. Functors are composed with the $\oplus$ operator:

$$\mathbf{data}\ (\oplus)\ f\ g\ a = Inl\ (f\ a) \mid Inr\ (g\ a)$$

$Fix\ (Arith_F \oplus Logic_F)$ represents a data type isomorphic to:

$$\mathbf{data}\ Exp = Lit\ Nat \mid Add\ Exp\ Exp$$
$$\qquad\qquad\ \mid If\ Exp\ Exp\ Exp \mid BLit\ Bool$$

Modular Mendler Algebras A type class is defined for every semantic function. For example, the evaluation function has the following class:

$$\mathbf{class}\ Eval\ f\ \mathbf{where}$$
$$\quad eval_{alg} :: Algebra_M\ f\ Value$$

In this class $eval_{alg}$ represents the evaluation algebra of a feature f.

Algebras of feature compositions are composed from the feature algebras:

$$\mathbf{instance}\ (Eval\ f, Eval\ g) \Rightarrow Eval\ (f \oplus g)\ \mathbf{where}$$
$$\quad eval_{alg}\ [\![\cdot]\!]\ (Inl\ fexp)\ = eval_{alg}\ [\![\cdot]\!]\ fexp$$
$$\quad eval_{alg}\ [\![\cdot]\!]\ (Inr\ gexp) = eval_{alg}\ [\![\cdot]\!]\ gexp$$

Overall evaluation is then defined as:

$$eval :: Eval\ f \Rightarrow Fix_M\ f \to Value$$
$$eval = fold_M\ eval_{alg}$$

Note that in order to avoid the repeated boilerplate of defining a new type class for every semantic function and corresponding instance for $\oplus$, MTC defines a single generic Coq type class, *FAlg*, that is indexed by the name of the semantic function. This class definition can be found in Figure 3 and subsumes all other algebra classes found in this paper. The paper uses more specific classes to make a gentler progression for the reader.

```
class f ≺: g where
  inj      :: f a → g a
  prj      :: g a → Maybe (f a)
  inj_prj :: prj ga = Just fa → ga = inj fa   -- law
  prj_inj :: prj ∘ inj = id                   -- law
instance (f ≺: g) ⇒ f ≺: (g ⊕ h) where
  inj fa         = Inl (inj fa)
  prj (Inl ga)  = prj ga
  prj (Inr ha) = Nothing
instance (f ≺: h) ⇒ f ≺: (g ⊕ h) where
  inj fa         = Inr (inj fa)
  prj (Inl ga)  = Nothing
  prj (Inr ha) = prj ha
instance f ≺: f where
  inj fa = fa
  prj fa = Just fa
```

Figure 1. Functor subtyping.

Injections and Projections of Functors Figure 1 shows the multi-parameter type class $\prec:$. This class provides a means to lift or inject (inj) (sub)functors f into larger compositions g and project (prj) them out again. The inj_prj and prj_inj laws relate the injection and projection methods in the $\prec:$ class, ensuring that the two are effectively inverses. The idea is to use the type class resolution mechanism to encode (coercive) subtyping between functors. In Coq this subtyping relation can be nicely expressed because Coq type classes [39] perform a backtracking search for matching instances. As such, highly overlapping definitions like the first and second instances are allowed. This is a notable difference to Haskell's type classes, which do not support backtracking. Hence, DTC's Haskell solution has to provide a biased choice that does not accurately model the expected subtyping relationship.

The in_f function builds a new term from the application of f to some subterms.

```
in_f :: f (Fix_M f) → Fix_M f
in_f fexp = λalg → alg (fold_M alg) fexp
```

The combination of in_f and inj allows us to define *smart constructors* such as:

```
inject :: (g ≺: f) ⇒ g (Fix_M f) → Fix_M f
inject gexp = in_f (inj gexp)
lit :: (Arith_F ≺: f) ⇒ Nat → Fix_M f
lit n = inject (Lit n)
blit :: (Logic_F ≺: f) ⇒ Bool → Fix_M f
blit b = inject (BLit b)

cond :: (Logic_F ≺: f)
     ⇒ Fix_M f → Fix_M f → Fix_M f → Fix_M f
cond c e_1 e_2 = inject (If c e_1 e_2)
```

Expressions are built with the smart constructors and used by operations like evaluation:

```
exp :: Fix_M (Arith_F ⊕ Logic_F)
exp = cond (blit True) (lit 3) (lit 2)
> eval exp
3
```

The out_f function exposes the toplevel functor again:

```
out_f :: Functor f ⇒ Fix_M f → f (Fix_M f)
out_f exp = fold_M (λrec fr → fmap (in_f ∘ rec) fr) exp
```

In combination with prj, we can pattern match on particular features.

```
project :: (g ≺: f, Functor f) ⇒
  Fix_M f → Maybe (g (Fix_M f))
project exp = prj (out_f exp)
isLit :: (Arith_F ≺: f, Functor f) ⇒ Fix_M f → Maybe Nat
isLit exp = case project exp of
  Just (Lit n) → Just n
  Nothing      → Nothing
```

2.6 Extensible Semantic Values

In addition to the modularity of language features, it is also desirable to have modular result types for semantic functions. For example, it is much cleaner to separate natural number and boolean values along the same lines as the $Arith_F$ and $Logic_F$ features. To easily achieve this extensibility, we make use of the same extensional encoding as for the expression language itself:

```
data NVal_F a = I Nat
data BVal_F a = B Bool
data Stuck_F a = Stuck
vi :: (NVal_F ≺: r) ⇒ Nat → Fix_M r
vi n = inject (I n)
vb :: (BVal_F ≺: r) ⇒ Bool → Fix_M r
vb b = inject (B b)
stuck :: (Stuck_F ≺: r) ⇒ Fix_M r
stuck = inject Stuck
```

Besides constructors for integer (vi) and boolean (vb) values, we also include a constructor denoting stuck evaluation ($stuck$).

To allow for an extensible return type r for evaluation, we need to parametrize the $Eval$ type class in r:

```
class Eval f r where
  eval_alg :: Algebra_M f (Fix_M r)
```

Now projection becomes essential for pattern matching on intermediate values:

```
instance (Stuck_F ≺: r, NVal_F ≺: r, Functor r) ⇒
  Eval Arith_F r where
    eval_alg ⟦·⟧ (Lit n)       = vi n
    eval_alg ⟦·⟧ (Add e_1 e_2) =
      case (project ⟦e_1⟧, project ⟦e_2⟧) of
        (Just (I n_1), (Just (I n_2))) → vi (n_1 + n_2)
        _                              → stuck
```

This concludes MTC's support for extensibility of data types and functions. To cater for meta-theory, MTC also addresses reasoning about these modular definitions.

3. Reasoning with Church Encodings

While Church encodings are the foundation of extensibility in MTC, Coq does not provide induction principles for them. It is an open problem to do so without resorting to axioms. MTC solves this problem with a novel axiom-free approach based on adaptations of two important aspects of folds discussed by Hutton [19].

3.1 The Problem of Church Encodings and Induction

Coq's own original approach [35] to inductive data types was based on Church encodings. It is well-known that Church encodings of

inductive data types have problems expressing induction principles such as A_{ind}, the induction principle for arithmetic expressions.

$$
\begin{aligned}
&A_{ind} :: \forall P :: (Arith \to Prop).\\
&\quad\ \forall H_l :: (\forall n.P\ (Lit\ n)).\\
&\quad\ \forall H_a :: (\forall a\ b.P\ a \to P\ b \to P\ (Add\ a\ b)).\\
&\quad\ \forall a.P\ a\\
&A_{ind}\ P\ H_l\ H_a\ e =\\
&\quad \mathbf{case}\ e\ \mathbf{of}\\
&\qquad Lit\ n \quad\ \to H_l\ n\\
&\qquad Add\ x\ y \to H_a\ a\ b\ (A_{ind}\ P\ H_l\ H_a\ x)\\
&\qquad\qquad\qquad\quad (A_{ind}\ P\ H_l\ H_a\ y)
\end{aligned}
$$

The original solution to this problem in Coq involved axioms for induction, which endangered strong normalization of the calculus (among other problems). This was the primary motivation for the creation of the *calculus of inductive constructions* [34] with built-in inductive data types.

Why exactly are proofs problematic for Church encodings, where inductive functions are not? After all, a Coq proof is essentially a function that builds a proof term by induction over a data type. Hence, the Church encoding should be able to express a proof as a fold with a *proof algebra* over the data type, in the same way it represents other functions.

The problem is that this approach severely restricts the propositions that can be proven. Folds over Church encodings are destructive and their result type cannot depend on the term being destructed. For example, it is impossible to express the proof for *type soundness* because it performs induction over the expression e mentioned in the type soundness property.

$$\forall e.\Gamma \vdash e : t \to \Gamma \vdash [\![e]\!] : t$$

This restriction is a showstopper for the semantics setting of this paper, as it rules out proofs for most (if not all) theorems of interest. Supporting relevant semantic reasoning requires a new approach that does not suffer from this restriction.

3.2 Type Dependency with Dependent Products

Hutton's first aspect of *folds* is that they become substantially more expressive with the help of tuples. The dependent products in Coq take this observation one step further. While a *fold* algebra cannot refer to the original term, it can simultaneously build a copy e of the original term and a proof that the property $P\ e$ holds for the new term. As the latter depends on the former, the result type of the algebra is a dependent product $\Sigma\ e.P\ e$. A generic algebra can exploit this expressivity to build a poor-man's induction principle, e.g., for the $Arith_F$ functor:

$$
\begin{aligned}
&A^2_{ind} :: \forall P :: (Fix_M\ Arith_F \to Prop).\\
&\quad\ \forall H_l :: (\forall n.P\ (lit\ n)).\\
&\quad\ \forall H_a :: (\forall a\ b.P\ a \to P\ b \to P\ (add\ a\ b)).\\
&\quad\ Algebra\ Arith_F\ (\Sigma\ e.P\ e)\\
&A^2_{ind}\ P\ H_l\ H_a\ e =\\
&\quad \mathbf{case}\ e\ \mathbf{of}\\
&\qquad Lit\ n \quad\ \to \exists\ (lit\ n)\ (H_l\ n)\\
&\qquad Add\ x\ y \to \exists\ (add\ (\pi_1\ x)\ (\pi_1\ y))\ (H_a\ (\pi_1\ x)\ (\pi_1\ y)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\pi_2\ x)\ (\pi_2\ y))
\end{aligned}
$$

Provided with the necessary proof cases, A^2_{ind} can build a specific proof algebra. The corresponding proof is simply a fold over a Church encoding using this proof algebra.

Note that since a proof is not a computational object, it makes more sense to use regular algebras than Mendler algebras. Fortunately, regular algebras are compatible with Mendler-based Church encodings as the following variant of $fold'_M$ shows.

$$
\begin{aligned}
&fold'_M :: Functor\ f \Rightarrow Algebra\ f\ a \to Fix_M\ f \to a\\
&fold'_M\ alg = fold_M\ (\lambda rec \to alg \circ fmap\ rec)
\end{aligned}
$$

3.3 Term Equality with the Universal Property

Of course, the dependent product approach does not directly prove a property of the original term. Instead, given a term, it builds a new term and a proof that the property holds for the new term. In order to draw conclusions about the original term from the result, the original and new term must be equal.

Clearly the equivalence does not hold for arbitrary terms that happen to match the type signatures $Fix_M\ f$ for Church encodings and $Algebra\ f\ (\Sigma\ e.P\ e)$ for proof algebras. Statically ensuring this equivalence requires additional *well-formedness conditions* on both. These conditions formally capture our notion of Church encodings and proofs algebras.

3.3.1 Well-Formed Proof Algebras

The first requirement, for algebras, states that the new term produced by application of the algebra is equal to the original term.

$$\forall alg :: Algebra\ f\ (\Sigma\ e.P\ e).\pi_1 \circ alg = in_f \circ fmap\ \pi_1$$

It is easy to verify that the A^2_{ind} proof algebra above satisfies this property. Thanks to A^2_{ind}'s parametric nature, there is no need for other proof algebras over $Arith_F$. Hence, in general, well-formedness needs to be proven only once for any data type and induction algebra.

3.3.2 Well-Formed Church Encodings

Well-formedness of algebras is not enough because a proof is not a single application of an algebra, but rather a $fold'_M$ of it. So the $fold'_M$ used to build a proof must be a *proper* $fold'_M$. As the Church encodings represent inductive data types as their folds, this boils down to ensuring that the Church encodings are well-formed.

Hutton's second aspect of folds formally characterizes the definition of a fold using its *universal property*:

$$h = fold'_M\ alg \Leftrightarrow h \circ in_f = alg\ h$$

In an initial algebra representation of an inductive data type, there is a single implementation of $fold'_M$ that can be checked once and for all for the universal property. In MTC's Church-encoding approach, every term of type $Fix_M\ f$ consists of a separate $fold'_M$ implementation that must satisfy the universal property. Note that this definition of the universal property is for a $fold'_M$ using a traditional algebra. As the only concern is the behavior of proof algebras (which are traditional algebras) folded over Church encodings, this is a sufficient characterization of well-formedness. Hinze [18] uses the same characterization for deriving Church numerals.

Fortunately, the left-to-right implication follows trivially from the definitions of $fold'_M$ and in_f, independent of the particular term of type $Fix_M\ f$. Hence, the only hard well-formedness requirement for Church-encoded terms e is that they satisfy the right-to-left implication of the universal property.

$$
\begin{aligned}
&\mathbf{type}\ UP\ f\ e =\\
&\quad \forall a\ (alg :: Algebra_M\ f\ a)\ (h :: Fix_M\ f \to a).\\
&\qquad (\forall e'.h\ (in_f\ e') = alg\ h\ e') \to h\ e = fold'_M\ alg\ e
\end{aligned}
$$

This property is easy to show for any given smart constructor. MTC actually goes one step further and redefines its smart constructors in terms of a new in_f, that only builds terms with the universal property:

$$in'_f :: Functor\ f \Rightarrow f\ (\Sigma\ e.UP\ f\ e) \to \Sigma\ e.UP\ f\ e$$

We constrain ourselves to reasoning about Church-encoded terms built from these smart-*er* constructors, as all of the nice properties

of initial algebras hold for these terms and, importantly, we have a handle on reasoning about them.

Two known consequences of the universal property are the famous *fusion* law, which describes the composition of a fold with another computation,

$$h \circ alg_1 = alg_2 \circ fmap\ h \ \Rightarrow\ h \circ fold'_M\ alg_1 = fold'_M\ alg_2$$

and the lesser known *reflection* law,

$$fold'_M\ in_f = id$$

3.3.3 Soundness of Input-Preserving Folds

Armed with the two well-formedness properties, we can now prove the key theorem that allows us to build inductive proofs over Church encodings:

Theorem 3.1. *Given a functor f, property P, and a well-formed P-proof algebra alg, for any Church-encoded f-term e with the universal property, we have that $P\ e$ holds.*

Proof. Given that $fold'_M\ alg\ e$ has type $\Sigma\ e'.P\ e'$, we have that $\pi_2\ (fold'_M\ alg\ e)$ is a proof for $P\ (\pi_1\ (fold'_M\ alg\ e))$. From that the lemma is derived as follows:

$$\begin{array}{ll} & P\ (\pi_1\ (fold'_M\ alg\ e)) \\ \Longrightarrow & \{\text{-well-founded algebra and fusion law -}\} \\ & P\ (fold'_M\ in_f\ e) \\ \Longleftrightarrow & \{\text{-reflection law -}\} \\ & P\ e \end{array}$$

□

Theorem 3.1 enables the construction of a statically-checked proof of correctness as a input-preserving fold of a proof algebra. This provides a means to achieve our true goal: modular proofs for extensible Church encodings.

4. Modular Proofs for Extensible Church Encodings

The aim of modularity in this setting is to first write a separate proof for every feature, and then compose the individual proofs into an overall proof for the feature composition. The feature proofs should be independent from one another, so that they can be reused for different feature compositions.

Fortunately, since proofs are essentially folds of proof algebras, all of the reuse tools developed in Section 2 apply here. In particular, composing proofs is a simple matter of combining the proof algebras with $\oplus$. Nevertheless, the transition to modular components does introduce several wrinkles in the reasoning process.

4.1 Algebra Delegation

Due to injection, propositions range over the abstract (super)functor f of the component composition. The signature of A^2_{ind}, for example, becomes:

$$\begin{array}{l} A^2_{ind} :: \forall f.Arith_F \prec: f \Rightarrow \\ \quad \forall P :: (Fix_M\ f \rightarrow Prop). \\ \quad \forall H_l :: (\forall n.P\ (lit\ n)). \\ \quad \forall H_a :: (\forall a\ b.P\ a \rightarrow P\ b \rightarrow P\ (add\ a\ b)). \\ \quad Algebra\ Arith_F\ (\Sigma\ e.P\ e) \end{array}$$

Consider building a proof of

$$\forall e.typeof\ e = Just\ nat \rightarrow \exists\ m :: nat.eval\ e = vi\ m$$

using A^2_{ind}. Then, the first proof obligation is

$$typeof\ (lit\ n) = Just\ nat \rightarrow \exists\ m :: nat.eval\ (lit\ n) = vi\ m$$

While this appears to follow immediately from the definition of $eval$, recall that $eval$ is a fold of an abstract algebra over f and is thus opaque. In order to proceed, we need an additional property on the behavior of this F-algebra, namely, that it delegates to the $Arith_F$-algebra, as expected:

$$\forall r\ (rec :: r \rightarrow Nat).eval_{alg}\ rec \circ inj = eval_{alg}\ rec$$

This delegation behavior follows from our approach: the intended structure of f is a $\oplus$-composition of features, and $\oplus$-algebras are intended to delegate to the feature algebras. We can formally capture the delegation behavior in a type class that serves as a precondition in our modular proofs.

$$\begin{array}{l} \textbf{class}\ (Eval\ f, Eval\ g, f \prec: g) \Rightarrow \\ \quad WF\_Eval\ f\ g\ \textbf{where} \\ \quad wf\_eval\_alg :: \forall r\ (rec :: r \rightarrow Nat)\ (e :: f\ r). \\ \quad\quad eval_{alg}\ rec\ (inj\ e :: g\ r) = \\ \quad\quad eval_{alg}\ rec\ e \end{array}$$

By providing the three instances of this class in Figure 2, one for each instance of $\prec:$, Coq can automatically build a proof of well-formedness alongside with the composite algebra.

$$\begin{array}{l} \textbf{instance}\ (Eval\ f, Eval\ g, Eval\ h, WF\_Eval\ f\ g) \Rightarrow \\ \quad WF\_Eval\ (f \prec: g \oplus h) \\ \textbf{instance}\ (Eval\ f, Eval\ g, Eval\ h, WF\_Eval\ f\ h) \Rightarrow \\ \quad WF\_Eval\ (f \prec: g \oplus h) \\ \textbf{instance}\ (Eval\ f) \Rightarrow WF\_Eval\ f\ f \end{array}$$

Figure 2. WF_Eval instances.

4.1.1 Automating Composition

A similar approach is used to automatically build the definitions and proofs of languages from pieces defined by individual features. In addition to functor and algebra composition, the framework derives several important reasoning principles as type class instances similarly to WF_Eval. These include the $DistinctSubFunctor$ class, which ensures that injections from two different subfunctors are distinct, and the $WF\_Functor$ class that ensures that $fmap$ distributes through injection.

Another example of automatic composition is the WF_Ind type class from Section 3.3.1, which enforces the well-formedness property on algebras. Because proofs of its $proj\_eq$ law have a very regular shape, they are actually easy to automate with Coq tactics. Because Coq's type class implementation is a variant of the $auto$ tactic, it is possible to tell the type class inferencer to execute a default proof script to build a WF_Ind instance on demand. If the script fails, Coq searches its instance database for custom proofs. However, none of the proof algebras from our case study need custom WF_Ind proofs; all are covered by the same proof script.

Figure 3 provides a summary of all the classes defined in MTC, noting whether the base instances of a particular class are provided by the user or inferred with a default instance. Importantly, instances of all these classes for feature compositions are built automatically, analogously to the instances in Figure 2.

4.2 Extensible Logical Relations

Most proofs appeal to rules of a logical relation that defines an important property. In Coq such logical relations are expressed as inductive data types of kind $Prop$. For instance, a soundness proof makes use of a judgment about the well-typing of values.

$$\begin{array}{l} \textbf{data}\ WTValue :: Value \rightarrow Type \rightarrow Prop\ \textbf{where} \\ \quad WTNat\ :: \forall n.WTValue\ (I\ n)\ TNat \\ \quad WTBool :: \forall b.WTValue\ (B\ b)\ TBool \end{array}$$

Class Definition	Description
class $Functor\ f$ **where** $fmap :: (a \to b) \to (f\ a \to f\ b)$ $fmap\_id :: fmap\ id = id$ $fmap\_fusion :: \forall g\ h.$ $fmap\ h \circ fmap\ g = fmap\ (h \circ g)$	**Functors** Supplied by the user
class $f \prec: g$ **where** $inj :: f\ a \to g\ a$ $prj :: g\ a \to Maybe\ (f\ a)$ $inj\_prj :: prj\ ga = Just\ fa \to$ $ga = inj\ fa$ $prj\_inj :: prj \circ inj = id$	Functor Subtyping Inferred
class $(Functor\ f, Functor\ g, f \prec: g) \Rightarrow$ $WF\_Functor\ f\ g$ **where** $wf\_functor :: \forall a\ b\ (h :: a \to b).$ $fmap\ h \circ inj = inj \circ fmap\ h$	Functor Delegation Inferred
class $(Functor\ h, f \prec: h, g \prec: h) \Rightarrow$ $DistinctSubFunctor\ f\ g\ h$ **where** $inj\_discriminate :: \forall a\ (fe :: f\ a)$ $(ge :: g\ a).inj\ fe \neq inj\ ge$	Functor Discrimination Inferred
class $FAlg\ Name\ t\ a\ f$ **where** $f\_algebra : Mixin\ T\ F\ A$	**Function Algebras** Supplied by the user
class $(f \prec: g, FAlg\ N\ t\ a\ g) \Rightarrow$ $WF\_FAlg\ (alg :: FAlg\ N\ t\ a\ g)$ **where** $wf\_algebra :: \forall rec\ (ft :: f\ t).$ $f\_algebra\ rec\ (inj\ fa) =$ $f\_algebra\ rec\ fa$	Algebra Delegation Inferred
class $PAlg\ Name\ f\ a$ **where** $p\_algebra :: Algebra\ f\ a$	**Proof Algebras** Supplied by the User
class $(Functor\ f, Functor\ g, f \prec: g) \Rightarrow$ $WF\_Ind\ g\ (alg :: PAlg\ N\ (\Sigma\ e.P\ e)\ f)$ **where** $proj\_eq :: \forall e.\pi_1\ (p\_algebra\ e) =$ $in_f\ (inj\ (fmap\ \pi_1\ e))$	Valid Proof Algebras Inferred

Figure 3. Type classes provided by MTC

When dealing with logical relations over extensible data types, the logical relations must be extensible as well. Extensibility of logical relations is obtained in much the same way as that of inductive data types: by means of Church encodings. The important difference is that logical relations are indexed data types; e.g., $WTValue$ is indexed by a value and a type. This requires functors indexed by values x of type i. For example, $WTNat_F\ v\ t$ is the corresponding indexed functor for the extensible variant of $WTNat$ above.

$$\begin{aligned}&\textbf{data}\ WTNat_F :: v \to t \to (WTV\ ::\ (v,t) \to Prop)\\ &\qquad\qquad\qquad\qquad\qquad\qquad \to (v,t) \to Prop\\ &\quad \textbf{where}\ WTNat :: \forall n.(NVal_F \prec: v, Functor\ v\\ &\qquad\qquad\qquad\qquad , NTyp_F \prec: t, Functor\ t)\\ &\qquad\qquad\qquad \Rightarrow WTNat_F\ v\ t\ WTV\ (vi\ n, tnat)\end{aligned}$$

The index here is a pair (v, t) of a value and a type. As object-language values and types are themselves extensible, the corresponding meta-language types v and t are parameters of the $WTNat$ functor.

To manipulate extensible logical relations, we need indexed algebras, fixpoints and operations:

$$\begin{aligned}&\textbf{type}\ iAlg\ i\ (f :: (i \to Prop) \to (i \to Prop))\ a\\ &\quad = \forall x :: i.f\ a\ x \to a\ x\\ &\textbf{type}\ iFix\ i\ (f :: (i \to Prop) \to (i \to Prop))\ (x :: i)\\ &\quad = \forall a :: i \to Prop.iAlg\ f\ a \to a\ x\ ...\end{aligned}$$

As these indexed variants are meant to construct logical relations, their parameters range over $Prop$ instead of Set. Fortunately, this shift obviates the need for universal properties for $iFix$-ed values: it does not matter *how* a logical relation is built, but simply that it exists. Analogues to $WF\_Functor$, $WF\_Algebra$, and $DistinctSubFunctor$ are similarly unnecessary.

4.3 Case Study: Soundness of an Arithmetic Language

Here we briefly illustrate modular reasoning with a case study proving soundness for the $Arith_F \oplus Logic_F$ language.

The previously defined $eval$ function captures the operational semantics of this language in a modular way and reduces an expression to a $NVal_F \oplus BVal_F \oplus Stuck_F$ value. Its type system is similarly captured by a modularly defined type-checking function $typeof$ that *maybe* returns a $TNat_F \oplus TBool_F$ type representation:

$$\begin{aligned}&\textbf{data}\ TNat_F\ t\ =\ TNat\\ &\textbf{data}\ TBool_F\ t = TBool\end{aligned}$$

For this language soundness is formulated as:

$$\begin{aligned}&\textbf{Theorem}\ soundness ::\\ &\forall e\ t\ env, typeof\ e = Just\ t \to WTValue\ (eval\ e\ env)\ t\end{aligned}$$

The proof of this theorem is a fold of a proof algebra over the expression e which delegates the different cases to separate proof algebras for the different features. A summary of the most noteworthy aspects of these proofs follows.

Sublemmas The modular setting requires that every case analysis is captured in a sublemma. This is necessary because the superfunctor is abstract, and hence the cases are not known locally; they have to be handled in a distributed fashion. Hence, modular lemmas built from proof algebras are not just an important tool for reuse in MTC – they are the main method of constructing extensible proofs.

Universal Properties Everywhere Universal properties are key to reasoning, and should thus be pervasively available throughout the framework. MTC has more infrastructure to support this.

As an example of their utility when constructing a proof, we may wish to prove a property of the extensible return value of an extensible function. Consider the $Logic_F$ case of the soundness proof: given that $typeof\ (If\ c\ e_1\ e_2) = Some\ t_1$, we wish to show that $WTValue\ (eval\ (If\ c\ e_1\ e_2))\ t_1$. If c evaluates to false, we need to show that $WTValue\ e_2\ t_1$.

Since $If\ c\ e_1\ e_2$ has type t_1, the definition of $typeof$ says that e_1 has type t_1:

$$\begin{aligned}&typeof_{alg}\ rec\ (If\ c\ e_1\ e_2) =\\ &\quad \textbf{case}\ project\ (rec\ c)\ \textbf{of}\\ &\qquad Just\ TBool\\ &\qquad\quad \to \textbf{case}\ (rec\ e_1, rec\ e_2)\ \textbf{of}\\ &\qquad\qquad\qquad (Just\ t_1, Just\ t_2) \to \textbf{if}\ eq_{type}\ t_1\ t_2\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ Just\ t_1\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ Nothing\\ &\qquad\qquad\qquad\quad\ \_ \qquad\qquad\qquad\quad \to Nothing\\ &\qquad Nothing \qquad \to Nothing\end{aligned}$$

In addition, the type equality test function, eq_{type}, says that e_1 and e_2 have the same type: $eq_{type}\ t_1\ t_2 = true$. We need to make use of a sublemma showing that $\forall t_1\ t_2.\ eq_{type}\ t_1\ t_2 = true \to t_1 = t_2$. As we have seen, in order to do so, the universal property must hold for $typeof\ \ e_1$. This is easily accomplished by packaging a proof of the universal property alongside t_1 in the $typeof$ function.

Using universal properties is so important to reasoning that this packaging should be the default behavior, even though it is computationally irrelevant. Thankfully, packaging becomes trivial with the use of smarter constructors. Moreover, these constructors have the additional advantage over standard smart constructors of being injective: $lit\ j = lit\ k \to j = k$, an important property for proving inversion lemmas. The proof of injectivity requires that

the subterms of the functor have the universal property, established by the use of in'_f. To facilitate this packaging, we provide a type synonym that can be used in lieu of Fix_M in semantic function signatures:

$$\textbf{type}\ UP_F\ f = Functor\ f \Rightarrow \Sigma\ e.(UP\ f\ e)$$

Furthermore, the universal property should hold for any value subject to proof algebras, so it is convenient to include the property in all proof algebras. MTC provides a predicate transformer, UP_P, that captures this and augments induction principles accordingly.

$$\begin{aligned}&UP_P :: Functor\ f \Rightarrow\\ &\quad (P :: \forall e.UP\ f\ e \to Prop) \to (e :: Fix_M\ f) \to \Sigma\ e.(P\ e)\end{aligned}$$

Equality and Universal Properties While packaging universal properties with terms enables reasoning, it does obfuscate equality of terms. In particular, two UP_F terms t and t' may share the same underlying term (i.e., $\pi_1\ t = \pi_1\ t'$), while their universal property proof component is different.[2]

This issue shows up in the definition of the typing judgment for values. This judgment needs to range over $UP_F\ f_v$ values and $UP_F\ f_t$ types (where f_v and f_t are the value and type functors), because we need to exploit the injectivity of *inject* in our inversion lemmas. However, knowing $WTValue\ v\ t$ and $\pi_1\ t = \pi_1\ t'$ no longer necessarily implies $WTValue\ v\ t'$ because t and t' may have distinct proof components. To solve this, we make use of two auxiliary lemmas $WTV_{\pi_1,v}$ and $WTV_{\pi_1,t}$ that establish the implication:

$$\begin{aligned}&\textbf{Theorem}\ WTV_{\pi_1,v}\ (i :: WTValue\ v\ t) =\\ &\quad \forall v'.\pi_1\ v = \pi_1\ v' \to WTValue\ v'\ t\end{aligned}$$

$$\begin{aligned}&\textbf{Theorem}\ WTV_{\pi_1,t}\ (i :: WTValue\ v\ t) =\\ &\quad \forall t'.\pi_1\ t' = \pi_1\ t' \to WTValue\ v\ t'\end{aligned}$$

Similar lemmas are used for other logical relations. Features which introduce new rules need to also provide proofs showing that they respect this "safe projection" property.

5. Higher-Order Features

Binders and general recursion are ubiquitous in programming languages, so MTC must support these sorts of higher-order features. The untyped lambda calculus demonstrates the challenges of implementing both these features with extensible church encodings.

5.1 Encoding Binders

To encode binders we use a *parametric HOAS* (PHOAS) [7] representation. PHOAS allows binders to be expressed as functors, while still preserving all the convenient properties of HOAS.

$Lambda_F$ is a PHOAS-based functor for a function feature with function application, abstraction and variables. The PHOAS style requires $Lambda_F$ to be parameterized in the type (v) of variables, in addition to the functor's usual type parameter r for recursive occurrences.

$$\textbf{data}\ Lambda_F\ v\ r = Var\ v \mid App\ r\ r \mid Lam\ (v \to r)$$

As before, smart constructors build extensible expressions:

$$\begin{aligned}&var :: (Lambda_F\ v \prec: f) \Rightarrow v \to Fix_M\ f\\ &var\ v = inject\ (Var\ v)\end{aligned}$$

$$\begin{aligned}&app :: (Lambda_F\ v \prec: f) \Rightarrow Fix_M\ f \to Fix_M\ f \to Fix_M\ f\\ &app\ e_1\ e_2 = inject\ (App\ e_1\ e_2)\end{aligned}$$

$$\begin{aligned}&lam :: (Lambda_F\ v \prec: f) \Rightarrow (v \to Fix_M\ f) \to Fix_M\ f\\ &lam\ f = inject\ (Lam\ f)\end{aligned}$$

[2] Actually, as proofs are usually opaque, we may not know whether they are equal or different.

5.2 Defining Non-Inductive Evaluation Algebras

Defining an evaluation algebra for the $Lambda_F$ feature presents additional challenges. Evaluation of the untyped lambda-calculus can produce a closure, requiring a richer value type than before:

$$\begin{aligned}&\textbf{data}\ Value =\\ &\quad Stuck \mid I\ Nat \mid B\ Bool \mid Clos\ (Value \to Value)\end{aligned}$$

Unfortunately, Coq does not allow such a definition, as the closure constructor is not strictly positive (recursive occurrences of $Value$ occur both at positive and negative positions). Instead, a closure is represented as an expression to be evaluated in the context of an environment of variable-value bindings. The environment is a list of values indexed by variables represented as natural numbers Nat.

$$\textbf{type}\ Env\ v = [v]$$

The modular functor $Closure_F$ integrates closure values into the framework of extensible values introduced in Section 2.6.

$$\begin{aligned}&\textbf{data}\ Closure_F\ f\ a = Clos\ (Fix_M\ f)\ (Env\ a)\\ &closure :: (Closure_F\ f \prec: r) \Rightarrow\\ &\qquad Fix_M\ f \to Env\ (Fix_M\ r) \to Fix_M\ r\\ &closure\ mf\ e = inject\ (Clos\ mf\ e)\end{aligned}$$

A first attempt at defining evaluation is:

$$\begin{aligned}&eval_{Lambda} :: \quad (Closure_F\ f \prec: r, Stuck_F \prec: r, Functor\ r) \Rightarrow\\ &\quad Algebra_M\ (Lambda_F\ Nat)\ (Env\ (Fix_M\ r) \to Fix_M\ r)\\ &eval_{Lambda}\ [\![\cdot]\!]\ exp\ env =\\ &\quad \textbf{case}\ exp\ \textbf{of}\\ &\qquad Var\ index \to env\ !!\ index\\ &\qquad Lam\ f \quad\ \to closure\ (f\ (length\ env))\ env\\ &\qquad App\ e_1\ e_2 \to\\ &\qquad\quad \textbf{case}\ project\ \$\ [\![e_1]\!]\ env\ \textbf{of}\\ &\qquad\qquad Just\ (Clos\ e_3\ env') \to [\![e_3]\!]\ ([\![e_2]\!]\ env : env')\\ &\qquad\qquad \_ \qquad\qquad\qquad\ \to stuck\end{aligned}$$

The function $eval_{Lambda}$ instantiates the type variable v of the $Lambda_F\ v$ functor with a natural number Nat, representing an index in the environment. The return type of the Mendler algebra is now a function that takes an environment as an argument. In the variable case there is an index that denotes the position of the variable in the environment, and $eval_{Lambda}$ simply looks up that index in the environment. In the lambda case $eval_{Lambda}$ builds a closure using f and the environment. Finally, in the application case, the expression e_1 is evaluated and analyzed. If that expression evaluates to a closure then the expression e_2 is evaluated and added to the closure's environment (env'), and the closure's expression e_3 is evaluated under this extended environment. Otherwise e_1 does not evaluate to a closure, and evaluation is stuck.

Unfortunately, this algebra is ill-typed on two accounts. Firstly, the lambda binder function f does not have the required type $Nat \to Fix_M\ f$. Instead, its type is $Nat \to r$, where r is universally quantified in the definition of the $Algebra_M$ algebra. Secondly, and symmetrically, in the App case, the closure expression e_3 has type $Fix_M\ f$ which does not conform to the type r expected by $[\![\cdot]\!]$ for the recursive call.

Both these symptoms have the same problem at their root. The Mendler algebra enforces inductive (structural) recursion by hiding that the type of the subterms is $Fix_M\ f$ using universal quantification over r. Yet this information is absolutely essential for evaluating the binder: we need to give up structural recursion and use general recursion instead. This is unsurprising, as an untyped lambda term can be non-terminating.

5.3 Non-Inductive Semantic Functions

Mixin algebras refine Mendler algebras with a more revealing type signature.

$$\textbf{type}\ \mathit{Mixin}\ t\ f\ a = (t \to a) \to f\ t \to a$$

This algebra specifies the type t of subterms, typically $Fix_M\ f$, the overall expression type. With this mixin algebra, $eval_{Lambda}$ is now well-typed:

$$\begin{aligned}&\mathit{eval}_{Lambda} :: (\mathit{Closure}_F\ e \prec: v, \mathit{Stuck}_F \prec: v) \Rightarrow\\ &\quad \mathit{Mixin}\ (\mathit{Fix}_M\ e)\ (\mathit{Lambda}_F\ \mathit{Nat})\\ &\quad (\mathit{Env}\ (\mathit{Fix}_M\ v) \to \mathit{Fix}_M\ v)\end{aligned}$$

Mixin algebras have an analogous implementation to $Eval$ as type classes, enabling all of MTC's previous composition techniques.

$$\begin{aligned}&\textbf{class}\ \mathit{Eval}_X\ f\ g\ r\ \textbf{where}\\ &\quad \mathit{eval}_{xalg} :: \mathit{Mixin}\ (\mathit{Fix}_M\ f)\ g\ (\mathit{Env}\ (\mathit{Fix}_M\ r) \to \mathit{Fix}_M\ r)\\ &\textbf{instance}\ (\mathit{Stuck}_F \prec: r, \mathit{Closure}_F\ f \prec: r, \mathit{Functor}\ r) \Rightarrow\\ &\qquad \mathit{Eval}_X\ f\ (\mathit{Lambda}_F\ \mathit{Nat})\ r\ \textbf{where}\\ &\quad \mathit{eval}_{xalg} = \mathit{eval}_{Lambda}\end{aligned}$$

Although the code of $eval_{Lambda}$ still looks suspiciously generally recursive, $eval_{Lambda}$ is actually *not* recursive because the recursive calls are abstracted as a parameter (like with Mendler algebras). As such, $eval_{Lambda}$ does not raise any issues with Coq's termination checker. Mixin algebras resemble the *open recursion* style which is used to model inheritance and mixins in object-oriented languages [9]. Still, Mendler encodings only accept Mendler algebras, so using mixin algebras with Mendler-style encodings requires a new form of fold.

In order to overcome the problem of general recursion, the open recursion of the mixin algebra is replaced with a bounded inductive fixpoint combinator, $boundedFix$, that returns a default value if the evaluation does not terminate after n recursion steps.

$$\begin{aligned}&\mathit{boundedFix} :: \forall f\ a.\mathit{Functor}\ f \Rightarrow \mathit{Nat} \to a \to\\ &\qquad \mathit{Mixin}\ (\mathit{Fix}_M\ f)\ f\ a \to \mathit{Fix}_M\ f \to a\\ &\mathit{boundedFix}\ n\ \mathit{def}\ \mathit{alg}\ e =\\ &\quad \textbf{case}\ n\ \textbf{of}\\ &\qquad 0 \to \mathit{def}\\ &\qquad m \to \mathit{alg}\ (\mathit{boundedFix}\ (m-1)\ \mathit{def}\ \mathit{alg})\ (\mathit{out}_f\ e)\end{aligned}$$

Note that the argument e is a Mendler-encoded expression of type $Fix_M\ f$. The key idea is to use out_f to unfold the expression once into a value of type $f\ \ (Fix_M\ \ f)$. $boundedFix$ then applies the algebra to that value recursively. In essence $boundedFix$ can define *generally* recursive operations by case analysis, since it can inspect values of the recursive occurrences. The use of the bound prevents non-termination.

Bounded Evaluation Evaluation can now be modularly defined as a bounded fixpoint of the mixin algebra $Eval_X$. The definition uses a distinguished bottom value, $\bot$, that represents a computation which does not finish within the given bound.

$$\begin{aligned}&\textbf{data}\ \ \bot_F\ \ a = \mathit{Bot}\\ &\ \bot = \mathit{inject}\ \mathit{Bot}\\ &\mathit{eval}_X :: (\mathit{Functor}\ f, \bot_F \prec: r, \mathit{Eval}_X\ f\ f\ r) \Rightarrow\\ &\quad \mathit{Fix}_M\ f \to \mathit{Fix}_M\ r\\ &\mathit{eval}_X\ e = \mathit{boundedFix}\ 1000\ (\backslash\_ \to \bot)\ \mathit{eval}_{xalg}\ e\ []\end{aligned}$$

5.4 Backwards compatibility

The higher-order PHOAS feature has introduced a twofold change to the algebras used by the evaluation function:

1. $eval_X$ uses mixin instead of Mendler algebras.

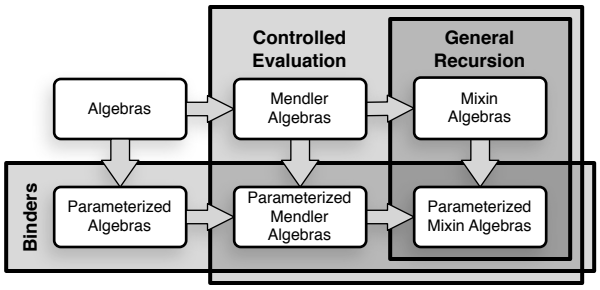

Figure 4. Hierarchy of Algebra Adaptation

2. $eval_X$ now expects algebras over a parameterized functor.

Fortunately, the first change is easily accommodated because Mendler algebras are compatible with mixin algebras. If a non-binder feature defines evaluation in terms of a Mendler algebra, it does not have to define a second mixin algebra to be used alongside binder features. The $mendlerToMixin$ function automatically derives the required mixin algebra from the Mendler algebra.

$$\begin{aligned}&\mathit{mendlerToMixin} :: \mathit{Algebra}_M\ f\ a \to \mathit{Mixin}\ (\mathit{Fix}_M\ g)\ f\ a\\ &\mathit{mendlerToMixin}\ \mathit{alg} = \mathit{alg}\end{aligned}$$

This conversion function can be used to adapt evaluation for the arithmetic feature to a mixin algebra:

$$\begin{aligned}&\textbf{instance}\ \mathit{Eval}\ \mathit{Arith}_F\ f \Rightarrow \mathit{Eval}_X\ f\ \mathit{Arith}_F\ r\ \textbf{where}\\ &\quad \mathit{eval}_{xalg}\ [\![\cdot]\!]\ e\ \mathit{env} =\\ &\qquad \mathit{mendlerToMixin}\ \mathit{evalAlgebra}\ (\mathit{flip}\ [\![\cdot]\!]\ \mathit{env})\ e\end{aligned}$$

The algebras of binder-free features can be similarly adapted to build an algebra over a parametric functor. Figure 4 summarizes the hierarchy of algebra adaptations. Non-parameterized Mendler algebras are the most flexible because they can be adapted and reused with both mixin algebras and parametric superfunctors. They should be used by default, only resorting to mixin algebras when necessary.

6. Reasoning with Higher-Order Features

The switch to a bounded evaluation function over parameterized Church encodings requires a new statement of soundness.

$$\begin{aligned}&\textbf{Theorem}\ \mathit{soundness}_X :: \forall f\ f_t\ \mathit{env}\ t\ \Gamma.\\ &\forall e_1 :: \mathit{Fix}_M\ (f\ (\mathit{Maybe}\ (\mathit{Fix}_M\ f_t)))\\ &\forall e_2 :: \mathit{Fix}_M\ (f\ \mathit{Nat}).\Gamma \vdash e_1 \equiv e_2 \to\\ &\mathit{typeof}\ e_1 = \mathit{Just}\ t \to \mathit{WTValue}\ (\mathit{eval}_X\ e_2\ \mathit{env})\ t\end{aligned}$$

The proof of $soundness_X$ features two substantial changes to the proof of $soundness$ from Section 4.3.

6.1 Proofs over Parametric Church Encodings

While, in its generality, $soundness_X$ mentions two distinct expressions e_1 and e_2, the toplevel invocation uses two instances of the same PHOAS expression $e :: \forall v.Fix_M\ (f\ v)$. They instantiate v with different types, according to the need of the typing and evaluation algebras.

In recursive invocations of $soundness_X$, that connection between e_1 and e_2 is no longer apparent, and as they have different types, Coq considers them to be distinct. Hence, case analysis on one does not convey information about the other. Chlipala [7] shows that the connection can be retained with the help of an auxiliary equivalence relation $\Gamma \vdash e_1 \equiv e_2$, where the environment Γ keeps track of the current variable bindings. The toplevel invocation, where the common origin of e_1 and e_2 is apparent, can easily supply a proof of this relation. By induction on this proof,

recursive invocations of $soundness_X$ can then analyze e_1 and e_2 in lockstep. Figure 5 shows the rules for determining equivalence of lambda expressions.

$$\frac{(x, x') \in \Gamma}{\Gamma \vdash var\ x \equiv var\ x'} \text{(EQV-VAR)} \qquad \frac{\Gamma \vdash e_1 \equiv e_1' \qquad \Gamma \vdash e_2 \equiv e_2'}{\Gamma \vdash app\ e_1\ e_2 \equiv app\ e_1'\ e_2'} \text{(EQV-APP)}$$

$$\frac{\forall x x'.(x, x'), \Gamma \vdash f(x) \equiv f'(x')}{\Gamma \vdash lam\ f \equiv lam\ f'} \text{(EQV-ABS)}$$

Figure 5. Lambda Equivalence Rules

6.2 Proofs for Non-Inductive Semantics Functions

Proofs for semantic functions that use $boundedFix$ proceed by induction on the bound. Hence, the reasoning principle for mixin-based bounded functions f is in general: provided a base case $\forall e, P(f\ 0\ e)$, and inductive case $\forall n\ e, (\forall e', P(f\ n\ e')) \rightarrow \forall e, P(f\ (n+1)\ e)$ hold, $\forall n\ e, P(f\ n\ e)$ also holds.

In the base case of $soundness_X$, the bound has been reached and $eval_X$ returns $\bot$. The proof of this case relies on adding to the $WTValue$ judgment the WF-BOT rule stating that every type is inhabited by $\bot$.

$$\frac{}{\vdash \bot : T} \text{(WF-BOT)}$$

Hence, whenever evaluation returns $\bot$, soundness trivially holds.

The inductive case is handled by a proof algebra whose statement includes the inductive hypothesis provided by the induction on the bound: $IH :: \forall n\ e, (\forall e', P(f\ n\ e')) \rightarrow P(f\ (n+1)\ e)$. The $App\ e_1\ e_2$ case of the soundness theorem illustrates the reason for including IH in the statement of the proof algebra. After using the induction hypothesis to show that $eval_X\ e_1\ env$ produces a well-formed closure $Clos\ e_3\ env'$, we must then show that evaluating e_3 under the $(eval_X\ e_2\ env) : env'$ environment is also well-formed. However, e_3 is not a subterm of $App\ e_1\ e_2$, so the conventional induction hypothesis for subterms does not apply. However, because $eval_X\ e_3\ ((eval_X\ e_2\ env) : env')$ is run with a smaller bound, the bounded induction hypothesis IH can be used.

6.3 Proliferation of Proof Algebras

In order to incorporate non-parametric inductive features in the $soundness_X$ proof, their existing proof algebras need to be adapted for the use in the above proof. In general, to cater for the four possible proof signatures of soundness, a naive approach requires four different proof algebras for an inductive non-parametric feature.[3] This is not acceptable, because reasoning about a feature's soundness should be independent of how a language adapts its evaluation algebra. Hence, MTC allows features to define a single proof algebra, and provides the means to adapt and reuse that proof algebra for the four variants. These proof algebra adaptations rely on *mediating type class instances* which automatically build an instance of the new proof algebra from the original proof algebra.

6.3.1 Adapting Proofs to Parametric Functors

Adapting a proof algebra over the expression functor to one over the indexed functor for the equivalence relation first requires a definition of equivalence for non-parametric functors. Fortunately, equivalence for any such functor f_{np} can be defined generically:

$$\frac{\Gamma \vdash \overline{a} \equiv \overline{b}}{\Gamma \vdash inject(C\ \overline{a}) \equiv inject(C\ \overline{b})} \text{(EQV-NP)}$$

[3] Introducing type-level binders would further compound the situation with four possible signatures for the $typeof$ algebra.

EQV-NP states that the same constructor C of f_{np}, applied to equivalent subterms $\overline{a}$ and $\overline{b}$, produces equivalent expressions.

The mediating type class adapts f_{np} proofs of propositions on two instances of the same PHOAS expression, like soundness, to proof algebras over the parametric functor.

$$\textbf{instance}\ (PAlg\ N\ P\ f_{np}) \Rightarrow iPAlg\ N\ P\ (\text{EQV-NP}\ f_{np})$$

This instance requires a small concession: proofs over f_{np} have to be stated more generally, in terms of two expressions with distinct superfunctors f and f' rather than two occurrences of the same expression. Note that such induction over two expressions requires a variant of WF_Ind for pairs of fixpoints.

6.3.2 Adapting Proofs to Non-Inductive Semantic Functions

To be usable regardless of whether $fold_M$ or $boundedFix$ is used to build the evaluation function, an inductive feature's proof needs to reason over an abstract fixpoint operator and induction principle. This is achieved by only considering a single step of the evaluation algebra and leaving the recursive call abstract:

$$\begin{aligned}
&\textbf{type}\ soundness\ e\ tp\ ev = \\
&\quad \forall env\ t.tp\ (out_f\ (\pi_1\ e)) = Just\ t \rightarrow \\
&\qquad WTValue\ (ev\ (out_t'\ (\pi_1\ e))\ env)\ t) \\
&\textbf{type}\ soundness_{alg}\ rec_t\ rec_e \\
&\quad (typeof_{alg} :: Mixin\ (Fix_M\ f)\ f\ (Maybe\ (Fix_M\ t)) \\
&\quad (eval_{alg} :: Mixin\ (Fix_M\ f)\ f\ (Env\ (Fix_M\ r) \rightarrow Fix_M\ r)) \\
&\quad (e :: Fix_M\ f)\ (e_UP' :: UP\ e) = \\
&\qquad \forall IHc :: (\forall e'. \\
&\qquad\quad soundness\ e'\ (typeof_{alg}\ rec_t)\ (eval_{alg}\ rec_e) \rightarrow \\
&\qquad\quad soundness\ e'\ rec_t\ rec_e). \\
&\qquad soundness\ e\ (typeof_{alg}\ rec_t)\ (eval_{alg}\ rec_e)
\end{aligned}$$

The hypothesis IHc is used to relate calls of rec_e and rec_t to applications of $eval_{alg}$ and $typeof_{alg}$.

A mediating type class instance again lifts proof algebras with this signature to one that includes the Induction Hypothesis generated by induction on the bound of $boundedFix$.

$$\textbf{instance}\ (PAlg\ N\ P\ E) \Rightarrow iPAlg\ N\ (IH \rightarrow P)\ E$$

7. Case Study

As a demonstration of the MTC framework, we have built a set of five reusable language features and combined them into a mini-ML [8] variant. The study also builds five other languages from these features.[4] Figure 6 presents the syntax of the expressions, values, and types provided by the features; each line is annotated with the corresponding feature.

The Coq files that implement these features average roughly 1100 LoC and come with a typing and evaluation function in addition to soundness and continuity proofs. Each language needs on average only 100 LoC to build its semantic functions and soundness proofs from the files implementing its features. The framework itself consists of about 2500 LoC.

The generic soundness proof, reused by each language, relies on a proof algebra to handle the case analysis of the main lemma. Each case is handled by a sub-algebra. These sub-algebras have their own set of proof algebras for case analysis or induction over an abstract superfunctor. The whole set of dependencies of a top-level proof algebra forms a *proof interface* that must be satisfied by any language which uses that algebra.

Such proof interfaces introduce the problem of *feature interactions* [4], well-known from modular component-based frameworks. In essence, a feature interaction is functionality (e.g., a function or

[4] Also available at `http://www.cs.utexas.edu/~bendy/MTC`

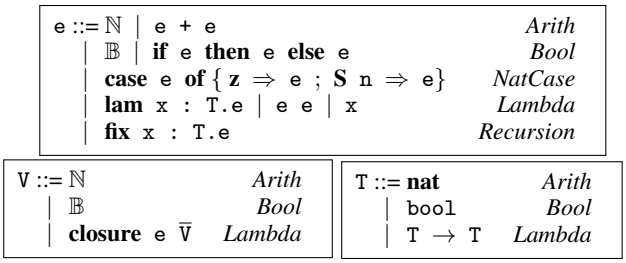

Figure 6. mini-ML expressions, values, and types

a proof) that is only necessary when two features are combined. An example from this study is the inversion lemma which states that values with type **nat** are natural numbers: $\vdash$ `x` : **nat** $\rightarrow$ `x` :: $\mathbb{N}$. The *Bool* feature introduces a new typing judgment, WT-BOOL for boolean values. Any language which includes both these features must have an instance of this inversion for WT-BOOL. Our modular approach supports feature interactions by capturing them in type classes. A missing case, like for WT-BOOL, can then be easily added as a new instance of that type class, without affecting or overriding existing code.

In the case study, feature interactions consist almost exclusively of inversion principles for judgments and the projection principles of Section 4.3. Thankfully, their proofs are relatively straightforward and can be dispatched by tactics hooked into the type class inference algorithm, analogously to the default instances for *WF_Ind* discussed in Section 3.3.1. These tactics help minimizing the number of interaction type class instances, which could otherwise easily grow exponentially in the number of features.

8. Related Work

This section discusses related work.

Modular Reasoning There is little work on mechanizing modular proofs for extensible components. An important contribution of our work is how to use *universal properties* to provide modular reasoning techniques for encodings of inductive data types that are compatible with theorem provers like Coq. Old versions of Coq, based on the *calculus of constructions* [10], also use Church encodings to model inductive data types [35]. However, the inductive principles to reason about those encodings had to be axiomatized and, among other problems, they endangered strong normalization of the calculus. The *calculus of inductive constructions* [34] has inductive data types built-in and was introduced to avoid the problems with Church encodings. In our work we go back to Church encodings to allow extensibility. However, we do not use standard induction principles since such inductive principles are themselves not extensible. Instead, by using a reasoning framework based on universal properties we get two benefits for the price of one: universal properties allow modular reasoning and they can be proved without axioms in Coq.

Extensibility Our approach to extensibility combines and extends ideas from existing solutions to the expression problem. The type class infrastructure for (Mendler-style) F-algebras is inspired by DTC [13, 40]. However type-level fixpoints, central to DTC, cannot be used in Coq because they require general recursion. To avoid general recursion, we use *least-fixpoints* encoded as Church encodings [5, 35]. Church encodings inspired other solutions to the expression problem before (especially in object-oriented languages) [30–32]. However those solutions do not use F-algebras: instead, they use an isomorphic representation called *object algebras* [31]. Object algebras are a better fit for languages where records are the main structuring construct (such as OO languages). Our solution differs from previous approaches in the use of Mendler-style F-algebras instead of conventional F-algebras or object algebras. Unlike previous solutions to the expression problem, which focus only on the extensibility aspects of implementations, we also deal with modular reasoning and the extensibility aspects of proofs and logical relations.

Mechanized Meta-Theory and Reuse Several ad-hoc tool-based approaches provide reuse, but none is based on a proof assistant's modularity features alone. The Tinkertype project [23] is a framework for modularly specifying formal languages. It was used to format the language variants used in Pierce's "Types and Programming Languages" [37], and to compose traditional pen-and-paper proofs.

Both Boite [6] and Mulhern [29] consider how to extend existing inductive definitions and reuse related proofs in the Coq proof assistant. Both their techniques rely on external tools that are no longer available and write extensions with respect to an existing specification. As such features cannot be checked independently or easily reused with new specifications. In contrast, our approach is fully implemented within Coq and allows for independent development and verification of features.

Delaware et al. [12] applied product-line techniques for modularizing mechanized meta-theory proofs. As a case study, they built type safety proofs for a family of extensions to Featherweight Java from a common base of features. Importantly, composition of these features was entirely manual, as opposed to the automated composition developed here.

Binding To minimize the work involved in modeling binders, MTC provides reusable binder components. The problem of modeling binders has received a lot of attention before. Certain proof assistants and type theories address this problem with better support for names and abstract syntax [36, 38]. In general-purpose proof assistants like Coq, however, such support is not available. A popular approach, widely used in Coq formalizations, is to use mechanization-friendly first-order representations of binders such as the *locally nameless* approach [1]. This involves developing a number of straightforward, but tedious infrastructure lemmas and definitions for each new language. Such tedious infrastructure can be automatically generated [2] or reused from data type-generic definitions [21]. However this typically requires additional tool support. A higher-order representation like PHOAS [7] avoids most infrastructure definitions. While we have developed PHOAS-based binders in MTC, first-order representations can be used as well.

Semantics and Interpreters While the majority of semantics formalization approaches use logical relations, we propose an approach based on interpreters. Of course, we are not the only ones to do so.

A particularly prominent line of work based on interpreters is that of using *monads* to structure semantics. Moggi [28] pioneered monads to model computation effects and structure denotation semantics. Liang et al. [25] introduced *monad transformers* to compose multiple monads and build modular interpreters. Jaskelioff et al. [20] used an approach similar to DTC in combination with monads to provide modular implementation of mathematical operational semantics. Our work could benefit of monads to model more complex language features. However, unlike previous work, we also have to consider modular reasoning. Monads introduce important challenges in terms of modular reasoning. Only very recently some modular proof techniques for reasoning about monads have been introduced [15, 33]. While this is a good step forward, it remains to be seen whether these techniques are sufficient to reason about suitably generalized modular statements like soundness.

The above approaches mainly involve pencil-&-paper proofs. Mechanization of interpreter-based semantics clearly poses its own challenges. Yet, it is highly relevant as it bestows the high degree of confidence in correctness directly on the executable artifact, rather than on an intermediate formulation based on logical relations. The only similar work in this direction, developed concurrently to our own, is that of Danielsson [11]. He uses the *partiality monad*, which fairly similar to our bounded fixpoint, to formalize semantic interpreters in Agda. He argues that this style is more easily understood and more obviously deterministic and computable than logical relations. Unlike us, Danielsson does not consider modularization of definitions and proofs.

9. Conclusion

Formalizing meta-theory can be very tedious. For larger programming languages the required amount of work can be overwhelming.

We propose a new approach to formalizing meta-theory that allows *modular* development of language formalizations. By building on existing solutions to modularity problems in conventional programming languages MTC allows modular definitions of semantic components. Furthermore, MTC also comes with modular reasoning techniques to reason about such modular semantic definitions. Our approach enables reuse of modular semantic components and proofs that deal with standard language constructs, and lets language designers focus on the interesting constructs of a language.

This paper addresses many, but obviously not all, of the fundamental issues for providing a formal approach to modular semantics. We will investigate further extensions of our approach, guided by the formalization of larger and more complex languages on top of our modular mini-ML variant. A particularly challenging issue we are aware of is the pervasive impact of new side-effecting features on existing definitions and proofs. We believe that existing work on modular *monadic* semantics [20, 24, 25] is a good starting point to overcome this hurdle.

References

[1] B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering Formal Metatheory. In *POPL '08*, 2008.

[2] B. E. Aydemir and S. Weirich. LNgen: Tool Support for Locally Nameless Representations, 2009. Unpublished manuscript.

[3] P. Bahr. Evaluation à la carte: Non-strict evaluation via compositional data types. In *Proceedings of the 23rd Nordic Workshop on Programming Theory*, NWPT '11, pages 38–40, 2011.

[4] D. Batory, J. Kim, and P. Höfner. Feature interactions, products, and composition. In *GPCE*, 2011.

[5] C. Böhm and A. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.*, 39, 1985.

[6] O. Boite. Proof reuse with extended inductive types. In *Theorem Proving in Higher Order Logics*, pages 50–65, 2004.

[7] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP'08*, 2008.

[8] D. Clément, T. Despeyroux, G. Kahn, and J. Despeyroux. A Simple Applicative Language: mini-ML. In *LFP '86*, 1986.

[9] W. R. Cook. *A denotational semantics of inheritance*. PhD thesis, Providence, RI, USA, 1989. AAI9002214.

[10] T. Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986.

[11] N. A. Danielsson. Operational semantics using the partiality monad, 2012. To appear at ICFP'12.

[12] B. Delaware, W. R. Cook, and D. Batory. Product lines of theorems. In *OOPSLA '11*, 2011.

[13] L. Duponcheel. Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters., 1995.

[14] B.E. Aydemir et. al. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *TPHOLs'05*, 2005.

[15] J. Gibbons and R. Hinze. Just do it: simple monadic equational reasoning. In *ICFP '11*, 2011.

[16] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24(1), Jan. 1977.

[17] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *ICFP '11*, 2011.

[18] R. Hinze. Church numerals, twice! *JFP*, 15(1):1–13, 2005.

[19] G. Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9(4):355–372, 1999.

[20] M. Jaskelioff, N. Ghani, and G. Hutton. Modularity and implementation of mathematical operational semantics. *Electron. Notes Theor. Comput. Sci.*, 229(5), March 2011.

[21] G. Lee, B. C. d. S. Oliveira, S. Cho, and K. Yi. Gmeta: A generic formal metatheory framework for first-order representations. In *ESOP 2012*, 2012.

[22] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 2009.

[23] M. Y. Levin and B. C. Pierce. Tinkertype: A language for playing with formal systems. *Journal of Functional Programming*, 13(2), March 2003.

[24] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *ESOP '96*, 1996.

[25] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL '95*, 1995.

[26] D. MacQueen. Modules for standard ML. In *LFP '84*, 1984.

[27] G. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen, September 1990.

[28] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1), July 1991.

[29] A. Mulhern. Proof weaving. In *WMM '06*, September 2006.

[30] B. C. d. S. Oliveira. Modular visitor components. In *ECOOP'09*, 2009.

[31] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses: Practical extensibility with object algebras. In *ECOOP'12*, 2012.

[32] B. C. d. S. Oliveira, R. Hinze, and A. Löh. Extensible and modular generics for the masses. In *Trends in Functional Programming*, 2006.

[33] B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. Effectiveadvice: disciplined advice with explicit effects. In *AOSD '10*, 2010.

[34] C. Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In *TLCA '93*, 1993.

[35] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the calculus of constructions. In *MFPS V*, 1990.

[36] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *CADE '99*, 1999.

[37] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[38] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

[39] M. Sozeau and N. Oury. First-class type classes. In *TPHOLs '08*, 2008.

[40] W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4), 2008.

[41] T. Uustalu and V. Vene. Coding recursion a la Mendler. In *WGP '00*, pages 69–85, 2000.

[42] P. Wadler. The Expression Problem. Email, November 1998. Discussion on the Java Genericity mailing list.

[43] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, pages 60–76, 1989.