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 closed 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-recursive 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 funcon \( f \). That function uses its type parameter \( a \) for inductive occurrences of the data type, leaving the data type definition open. \( \text{Arith}_F \) is an example funcon for a simple language of arithmetic expressions with literals and addition.

\[
\text{data } \text{Arith}_F \ a = \text{Lit } \text{Nat} \mid \text{Add } \ a \ a
\]

The explicitly recursive definition \( \text{Fix} \ f \) closes the open recursion of a funcon \( f \).

\[
\text{data } \text{Fix}_{\text{DTC}} \ f = \text{In } (f (\text{Fix}_{\text{DTC}} \ f))
\]

Applying \( \text{Fix}_{\text{DTC}} \) to \( \text{Arith}_F \) builds the data type for arithmetic expressions.

\[
\text{type } \text{Arith} = \text{Fix}_{\text{DTC}} \text{Arith}_F
\]

Functions over \( \text{Fix}_{\text{DTC}} \ f \) are expressed as folds of \( F \)-Algebras.

\[
\text{type } \text{Arith} \ a = f \ a \ a
\]

\[
\text{fold}_{\text{DTC}} \ f \Rightarrow \text{Algebra } f \ a \Rightarrow \text{Fix}_{\text{DTC}} \ f \ a
\]

\[
\text{fold}_{\text{DTC}} \ f a = \text{alg} (f (\text{fold}_{\text{DTC}} \ f) a)
\]

For example, the evaluation algebra of \( \text{Arith}_F \) is defined as:

\[
\text{data } \text{Value} = \text{I } \text{Int} \mid \text{B } \text{Bool}
\]

\[
\text{eval}_{\text{Arith}} :: \text{Algebra } \text{Arith}_F \text{Value}
\]

\[
\text{eval}_{\text{Arith}} \text{(Lit } n) = \text{I } n
\]

\[
\text{eval}_{\text{Arith}} (\text{Add } (\text{I } v_1) (\text{I } v_2)) = \text{I } (v_1 + v_2)
\]

Note that the recursive occurrences in \( \text{eval}_{\text{Arith}} \) are of the same type as the result type \( \text{Value} \). 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:

\[
\begin{align*}
\text{eval}_{\text{Arith}} : & \text{Fix}_{\text{DTC}} \ A \\
\text{eval}_{\text{Arith}} : & \text{fold}_{\text{DTC}} \text{eval}_{\text{Arith}} \\
\text{eval}_{\text{Arith}} : & \text{eval}_{\text{Arith}} (\text{Add } (\text{Lit } 1) (\text{Lit } 2))
\end{align*}
\]

Unfortunately, DTC’s double use of general recursion is not allowed by Coq. Coq does not accept the type-level fixpoint combinator \( \text{Fix}_{\text{DTC}} \) because it is not strictly positive. The \( \text{fold}_{\text{DTC}} \) 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:

\[
\text{type } \text{Fix} \ f = \forall a. \text{Algebra } f \ a \ a
\]

\[
\text{fold} :: \text{Algebra } f \ a \ a \Rightarrow \text{Fix} \ f \ a \ a
\]

\[
\text{fold} \text{alg} = \text{fa} \text{alg}
\]

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 \( \text{fold} \) is trivial: it applies the data type fold \( \text{Fix} \) to the algebra.

\[
\text{Boolean values are not needed yet, but they will be useful later in this section.}
\]
For example, these are the Church encodings of \textit{Arith}_F’s literals and addition:

\begin{align*}
\text{lit} : & : \text{Nat} \rightarrow \text{Fix Arith}_F \\
\text{lit } n & = \lambda a \text{lg} \rightarrow a \text{ (Lit } n) \\
\text{add} : & : \text{Fix Arith}_F \rightarrow \text{Fix Arith}_F \rightarrow \text{Fix Arith}_F \\
\text{add } e_1 \ e_2 & = \lambda a \text{lg} \rightarrow (\text{Add } (\text{fold } a \text{lg } e_1) (\text{fold } a \text{lg } e_2))
\end{align*}

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:

\[ [\text{if } e_1 \ e_2 \ e_3] = \begin{cases} 
\text{true} & \text{if } e_1 \\
\text{false} & \text{if } e_2 \\
\text{true} & \text{if } e_3
\end{cases} \]

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

\begin{verbatim}
data Logic \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a \_a }
class f <-: g where
  inj :: f a → g a
  proj :: g a → Maybe (f a)
  inj_prj :: proj ga = Just fa → proj = inj fa -- law
  prj_inj :: proj ∘ inj = id -- law

instance (f <-: g) ⇒ f <-: (g ⊕ h) where
  inj fa = Inl (inj fa)
  proj (Inl ga) = proj ga
  proj (Inr ha) = Nothing

instance (f <-: h) ⇒ f <-: (g ⊕ h) where
  inj fa = Inr (inj fa)
  proj (Inl ga) = Nothing
  proj (Inr ha) = prj ha

instance f <-: f where
  inj fa = fa
  proj fa = Just fa

Figure 1. Functor subtyping.

Injections and Projections of Functors Figure 1 shows the multi-parameter type class <-:. This class provides a means to lift or inject (inj) (sub)functors f into larger compositions g and project (proj) them out again. The inj_prj and proj_inj laws relate the injection and projection methods in the <-: 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 inj function builds a new term from the application of f to some subterms.

\[ \text{inj} :: f \ (\text{Fix}_M \ f) \rightarrow \text{Fix}_M \ f \]
\[ \text{inj} \ \text{fexp} = \text{fold}_M \ (\text{inj} \ \text{fexp}) \]

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

\[ \text{inject} :: (g <-: f) ⇒ g \ (\text{Fix}_M \ f) \rightarrow \text{Fix}_M \ f \]
\[ \text{inject} \ \text{gexp} = \text{inj} \ (\text{inject} \ \text{gexp}) \]
\[ \text{lit} :: (\text{Arith}_F <-: f) ⇒ \text{Nat} \rightarrow \text{Fix}_M \ f \]
\[ \text{lit} \ n \ = \text{inject} \ (\text{Lit} \ n) \]
\[ \text{blit} :: (\text{Logic}_F <-: f) ⇒ \text{Bool} \rightarrow \text{Fix}_M \ f \]
\[ \text{blit} \ b \ = \text{inject} \ (\text{BLit} \ b) \]
\[ \text{cond} :: (\text{Logic}_F <-: f) ⇒ \text{Fix}_M \ f \rightarrow \text{Fix}_M \ f \rightarrow \text{Fix}_M \ f \]
\[ \text{cond} \ c \ e_1 \ e_2 \ = \text{inject} \ (\text{If} \ c \ e_1 \ e_2) \]

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

\[ \text{exp} :: \text{Fix}_M \ (\text{Arith}_F \oplus \text{Logic}_F) \]
\[ \text{exp} \ = \text{cond} \ (\text{blit} \ True) \ (\text{lit} \ 3) \ (\text{lit} \ 2) \]
\[ > \text{eval} \ \text{exp} \]
\[ = 3 \]

The \text{outf} function exposes the toplevel functor again:

\[ \text{outf} :: \text{Functor} \ f ⇒ \text{Fix}_M \ f \rightarrow f \ (\text{Fix}_M \ f) \]
\[ \text{outf} \ \text{exp} = \text{fold}_M \ (\lambda \text{rec} \ f r \rightarrow \text{fmap} \ (\text{inj} \circ \text{rec}) \ f \ \text{exp}) \]

In combination with \text{proj}, we can pattern match on particular features.

\[ \text{project} :: (g <-: f, \text{Functor} \ f) ⇒ \text{Fix}_M \ f \rightarrow \text{Maybe} \ (g \ (\text{Fix}_M \ f)) \]
\[ \text{project} \ \text{exp} = \text{proj} \ (\text{outf} \ \text{exp}) \]
\[ \text{isLit} :: (\text{Arith}_F <-: f, \text{Functor} \ f) ⇒ \text{Fix}_M \ f \rightarrow \text{Maybe} \ \text{Nat} \]
\[ \text{isLit} \ \text{exp} = \text{case} \ \text{project} \ \text{exp} \ of \]
\[ \text{Just} \ (\text{Lit} \ n) \ → \text{Just} \ n \]
\[ \text{Nothing} \ → \text{Nothing} \]

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 \text{Arith}_F and \text{Logic}_F features. To easily achieve this extensibility, we make use of the same extensional encoding as for the expression language itself:

\[ \text{data} \ \text{NVal}_F \ a = \text{I} \ \text{Nat} \]
\[ \text{data} \ \text{BVal}_F \ a = \text{B} \ \text{Bool} \]
\[ \text{data} \ \text{Stick}_F \ a = \text{Stick} \]
\[ \text{vi} :: (\text{NVal}_F <-: r) ⇒ \text{Nat} \rightarrow \text{Fix}_M \ r \]
\[ \text{vb} :: (\text{BVal}_F <-: r) ⇒ \text{Bool} \rightarrow \text{Fix}_M \ r \]
\[ \text{vb} \ b \ = \text{inject} \ (\text{B} \ b) \]
\[ \text{stick} :: (\text{Stick}_F <-: r) ⇒ \text{Fix}_M \ r \]
\[ \text{stick} \ = \text{inject} \ \text{Stick} \]

Besides constructors for integer (\text{vi}) and boolean (\text{vb}) values, we also include a destructor denoting stuck evaluation (\text{stick}).

To allow for an extensible return type \text{r} for evaluation, we need to parametrize the \text{Eval} type class in \text{r}:

\[ \text{class} \ \text{Eval} \ f \ r \ where \]
\[ \text{eval}_{\text{alg}} :: \text{Algebra}_M \ f \ (\text{Fix}_M \ r) \]

Now projection becomes essential for pattern matching on intermediate values:

\[ \text{instance} (\text{Stick}_F <-: r, \text{NVal}_F <-: r, \text{Functor} \ r) ⇒ \]
\[ \text{Eval} \ \text{Arith}_F \ r \ where \]
\[ \text{eval}_{\text{alg}} \ [\text{Lit} \ n] = \text{vi} \ n \]
\[ \text{eval}_{\text{alg}} \ [\text{Add} \ e_1 \ e_2] = \]
\[ \text{case} \ (\text{project} \ [e_1], \text{project} \ [e_2]) \ of \]
\[ (\text{Just} \ (\text{I} \ n_1), (\text{Just} \ (\text{I} \ n_2))) → \text{vi} \ (n_1 + n_2) \]
\[ − → \text{stick} \]

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.

$$A_{ind} :: \forall P :: (\text{Arith} \to \text{Prop}).$$

$$\forall H :: (\forall n. P (\text{Lit } n)).$$

$$\forall H :: (\forall a b. P a \to P b \to P (\text{Add } a b)).$$

$$\forall a. P a$$

$$A_{ind} P H : H a e =$$

case $e$ of

- Lit $n$ $\to H : H n$
- Add $x y \to H : H a b (A_{ind} P H : H a x) (A_{ind} P H : H b y)$

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 the 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 cannot refer to the original term, it can simultaneously build a copy of the original term and a proof that the property $P$ holds for the new term. As the latter depends on the former, the result type of the original term and a proof that the property

$$P e$$

$$\text{true}$$

holds for the universal property. Instead, given a term, it builds a property of the original term. Instead, given a term, it builds a fold using its definition of a fold using its universal property:

$$\text{fold}_{\text{M}} : \text{Functor } f \Rightarrow \text{Algebra } f a \Rightarrow \text{Fix}_{\text{M}} f f \Rightarrow a$$

$$\text{fold}_{\text{M}} alg = \text{fold}_{\text{M}} (\lambda \text{rec} \Rightarrow \text{alg} \circ \text{fmap} \text{ rec})$$

### 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 $\text{Fix}_{\text{M}} f$ for Church encodings and $\text{Algebra } f (\Sigma e. P e)$ 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.

$$\forall alg :: \text{Algebra } f (\Sigma e. P e) \Rightarrow \pi_2 \circ alg \Rightarrow \pi_1 \circ \text{fmap} \pi_1$$

It is easy to verify that the $A_{ind}$ proof algebra above satisfies this property. Thanks to $A_{ind}$’s parametric nature, there is no need for other proof algebras over $\text{Arith}_F$. 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 $\text{fold}_{\text{M}}$ of it. So the $\text{fold}_{\text{M}}$ used to build a proof must be a proper $\text{fold}_{\text{M}}$. 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:

$$h = \text{fold}_{\text{M}} alg \Leftrightarrow h \circ \text{in}_{\text{F}} = \text{alg } h$$

In an initial algebra representation of an inductive data type, there is a single implementation of $\text{fold}_{\text{M}}$ that can be checked once and for all for the universal property. In MTC’s Church-encoding approach, every term of type $\text{Fix}_{\text{M}} f$ consists of a separate $\text{fold}_{\text{M}}$ implementation that must satisfy the universal property. Note that this definition of the universal property is for a $\text{fold}_{\text{M}}$ 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 $\text{fold}_{\text{M}}$ and $\text{in}_{\text{F}}$, independent of the particular term of type $\text{Fix}_{\text{M}} f$. Hence, the only hard well-formedness requirement for Church-encoded terms is that they satisfy the right-to-left implication of the universal property.

$$\text{type } UP f e =$$

$$\forall a ([\text{alg :: } \text{Algebra}_F f a]) (h :: \text{Fix}_F f f \Rightarrow a).$$

$$\forall e'. (h \circ \text{in}_{\text{F}} e') = \text{alg } h e' \Rightarrow h e = \text{fold}_{\text{M}} alg e$$

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 $\text{in}_{\text{F}}$, that only builds terms with the universal property:

$$\text{in}_{\text{F}} :: \text{Functor } f \Rightarrow f (\Sigma e. UP f e) \Rightarrow \Sigma e. UP f e$$

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,

\[ h \circ \text{fold}_M \circ f \mapsto h \circ \text{fold}_M \circ f \circ \text{fold}_M \circ g \quad \text{and the lesser known reflection law,} \]

\[ \text{fold}_M \circ \text{in}_f = \text{id} \]

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 \( \text{alg} \), for any Church-encoded \( f \)-term \( e \) with the universal property, we have that \( P \circ e \) holds.

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

\[ P (\pi_1 (\text{fold}_M \circ \text{alg} \circ e)) \quad \Rightarrow \quad \{\text{well-formed algebra and fusion law} \} \]

\[ P (\text{fold}_M \circ \text{in}_f \circ e) \quad \Leftrightarrow \quad \{\text{reflection law} \} \]

\[ P \circ e \quad \square \]

Theorem 3.1 enables the construction of a statically-checked proof of correctness as an 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 \( A_{\text{mod}} \), for example, becomes:

\[ A_{\text{mod}} :: \forall f. \text{Arith}_F \circ f \circ \text{prop} \Rightarrow \]

\[ \forall P :: (\text{Fix}_M \circ f \rightarrow \text{prop}), \]

\[ \forall H :: (\forall \text{ap} P \circ \text{lit} n), \]

\[ \forall \text{He} :: (\forall a \circ P \circ a \rightarrow P \circ b \rightarrow P (\text{add} a b)). \]

**Algebra Arith\_F (\Sigma e.P \circ e)\]

Consider building a proof of

\[ \forall e. \text{typeof} e = \text{Just nat} \rightarrow \exists m :: \text{nat.eval} e = vi m \]

using \( A_{\text{mod}} \). Then, the first proof obligation is

\[ \text{typeof} \circ (\text{lit} n) = \text{Just nat} \rightarrow \exists m :: \text{nat.eval} \circ (\text{lit} n) = vi m \]

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:

\[ \forall r :: (\text{rec} :: r \rightarrow \text{Nat}) \circ (e :: f r) = \text{eval}_\text{alg} \circ (\text{inj} e :: g r) \]

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

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.

**data** \( \text{WTValue} :: \text{Value} \rightarrow \text{Type} \rightarrow \text{prop} \)

**where** \( \text{WTNat} :: \forall n. \text{WTValue} \circ (\text{I n}) \circ \text{TNat} \)

**WTBool :: \forall b. \text{WTValue} \circ (\text{B b}) \circ \text{TBool} \)

---

**Figure 2.** WF\_Eval instances.
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 $WTValue$.

Fortunately, for proving inversion lemmas. The proof of injectivity requires 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 $VTW t_1 t_2$. $eq_{type} t_1 t_2 = true \rightarrow t_1 = t_2$. 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: $lit j = lit k \rightarrow j = k$, 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 \( \text{in} \). To facilitate this packaging, we provide a type synonym that can be used in lieu of \( \text{Fix} \) in semantic function signatures:

\[
\text{type}\; UP F = \text{Functor} f \Rightarrow \Sigma e.(UP f e)
\]

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, \( UP F \), that captures this and augments induction principles accordingly.

\[
UP F :: \text{Functor} f \Rightarrow (P :: \forall e.UP f e \rightarrow \text{Prop}) \rightarrow (e :: \text{Fix} M f) \rightarrow \Sigma e.(P e)
\]

### 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 component is different.\(^2\)

This issue shows up in the definition of the typing judgment for values. This judgment needs to range over all proof components. \( UP F \) is different.

\[
\text{as before, smart constructors build extensible expressions:}
\]

\[
\text{in addition to the functor's usual type parameter requires still preserving all the convenient properties of HOAS.}
\]

To encode binders we use a complementing both these features with extensible church encodings.

### Binders and General Recursion

Are ubiquitous in programming languages, so MTC must support these sorts of higher-order features.

### 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 functions, while still preserving all the convenient properties of HOAS. \( \Lambda \) is a PHOAS-based functor for a function feature with function application, abstraction and variables. The PHOAS style requires \( \Lambda \) to be parameterized in the type (\( v \)) of variables, in addition to the functor's usual type parameter \( r \) for recursive occurrences.

\[
\text{data}\; \Lambda \vdash v \vdash r = \text{Var} v | \text{App} r r | \text{Lam} (v \rightarrow r)
\]

As before, smart constructors build extensible expressions:

\[
\begin{align*}
\text{var} :: (\Lambda \vdash v \vdash f) & \Rightarrow v \rightarrow \text{Fix} M f \\
\text{var} v & = \text{inject} (\text{Var} v) \\
\text{app} :: (\Lambda \vdash v \vdash f) & \Rightarrow \text{Fix} M f \rightarrow \text{Fix} M f \rightarrow \text{Fix} M f \\
\text{app} e_1 e_2 & = \text{inject} (\text{App} e_1 e_2) \\
\text{lam} :: (\Lambda \vdash v \vdash f) & \Rightarrow (v \rightarrow \text{Fix} M f) \rightarrow \text{Fix} M f \\
\text{lam} f & = \text{inject} (\text{Lam} f)
\end{align*}
\]

\(^2\) 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 \( \Lambda \) feature presents additional challenges. Evaluation of the untyped lambda-calculus can produce a closure, requiring a richer value type than before:

\[
\text{data Value} = \text{Stuck} | I \text{ Nat} | B \text{ Bool} | \text{Clos} (\text{Value} \rightarrow \text{Value})
\]

Unfortunately, Coq does not allow such a definition, as the closure constructor is not strictly positive (recursive occurrences of \( \text{Value} \) 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 \( \text{Nat} \).

\[
\text{type}\; Env v = [v]
\]

The modular functor \( \text{Closure} F \) integrates closure values into the framework of extensible values introduced in Section 2.6.

\[
\begin{align*}
\text{data}\; \text{Closure} F f a & = \text{Clos} (\text{Fix} M f) (\text{Env} a) \\
\text{closure} :: (\text{Closure} F f \rightarrow r) & \Rightarrow \text{Fix} M f \rightarrow \text{Env} (\text{Fix} M r) \rightarrow \text{Fix} M r \\
\text{closure mf} e & = \text{inject} (\text{Clos mf e})
\end{align*}
\]

A first attempt at defining evaluation is:

\[
\begin{align*}
\text{evalLambda} :: & \quad (\text{Closure} F f \rightarrow r, \text{Stuck} \rightarrow r, \text{Functor} r) \Rightarrow \\
& \quad \text{Algebra } \Lambda (\text{Lambda} F \vdash \text{Nat}) (\text{Env} (\text{Fix} M r) \rightarrow \text{Fix} M r) \\
\text{evalLambda} (\text{[]} \exp env) & = \\
\text{case}\; \exp\;\text{of} & \quad \text{Var index} \rightarrow \exp env!!\text{index} \\
& \quad \text{Lam} \; v \rightarrow \text{closure} (f \langle\text{length}\;\text{env}\rangle)\;\exp env \\
& \quad \text{App} \; e_1 \; e_2 \rightarrow \\
& \quad \text{case}\; \text{project}\; 8\;e_3\;\exp\;\text{of} \\
& \quad \text{Just} \; (\text{Clos} \; e_5 \; \exp env') \rightarrow \exp e_3 ([\exp e_2] \exp env : \exp env') \\
& \quad \rightarrow \text{stuck}
\end{align*}
\]

The function \( \text{evalLambda} \) instantiates the type variable \( v \) of the \( \Lambda \) \( v \) functor with a natural number \( \text{Nat} \), representing an index in the environment. The return type of the \( \Lambda \) 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 \( \text{evalLambda} \) simply looks up that index in the environment. In the lambda case \( \text{evalLambda} \) 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 (\( \exp env' \)), 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 \( \text{Nat} \rightarrow \text{Fix} M f \). Instead, its type is \( \text{Nat} \rightarrow r \), where \( r \) is universally quantified in the definition of the \( \text{Algebra} \text{M} \) algebra. Secondly, and symmetrically, in the \( \text{App} \) case, the closure expression \( e_3 \) has type \( \text{Fix} M f \) which does not conform to the type \( r \) expected by \( \text{proj} \) for the recursive call.

Both these symptoms have the same problem at their root. The \( \Lambda \) algebra enforces inductive (structural) recursion by hiding that the type of the subterms is \( \text{Fix} M f \) 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.

\[
\text{type Mixin } t f a = (t \to a) \to f t \to a
\]

This expression specifies the type \( t \) of subterms, typically \( \text{Fix}_M f \), the overall expression type. With this mixin algebra, \( \text{eval}_{\text{Mix}} \) is now well-typed:

\[
\begin{align*}
\text{eval}_{\text{Mix}} : \text{Closure}_P e \prec v, \text{Stuck}_P \prec v \Rightarrow \\
\text{Mendler } (\text{Fix}_M e) \text{ (Lambda}_F \text{ Nat)} \\
(\text{Env } (\text{Fix}_M v) \to \text{Fix}_M v)
\end{align*}
\]

Mixin algebras have an analogous implementation to \( \text{Eval} \) as type classes, enabling all of MTC’s previous composition techniques.

\[
\text{class EvalX } f g r \text{ where} \\
\text{eval}_{\text{Mix}} (\text{Fix}_M f) g (\text{Env } (\text{Fix}_M r) \to \text{Fix}_M r)
\]

\[
\text{instance } (\text{Stuck}_P \prec r, \text{Closure}_P f \prec r, \text{Function } r) \Rightarrow \text{EvalX } f (\text{Lambda}_F \text{ Nat } r) \text{ where} \\
\text{eval}_{\text{Mix}} = \text{eval}_{\text{Mix}}
\]

Although the code of \( \text{eval}_{\text{Mix}} \) still looks suspiciously generally recursive, \( \text{eval}_{\text{Mix}} \) is actually not recursive because the recursive calls are abstracted as a parameter (like with Mendler algebras). As such, \( \text{eval}_{\text{Mix}} \) 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, \( \text{boundedFix} \), that returns a default value if the evaluation does not terminate after \( n \) recursion steps.

\[
\begin{align*}
\text{boundedFix } : \forall a. \text{Function } f \to \text{Nat } \to a \to \text{Fix}_M f \to a \\
\text{boundedFix } n \text{ def alg } e = \\
\text{case } n \text{ of} \\
0 \Rightarrow \text{def} \\
(m \Rightarrow \text{alg } (\text{boundedFix } (m - 1) \text{ def alg}) \text{ (outI } e))
\end{align*}
\]

Note that the argument \( e \) is a Mendler-encoded expression of type \( \text{Fix}_M f \). The key idea is to use \( \text{outI } e \) to unfold the expression once into a value of type \( f \text{ (Fix}_M f) \), \( \text{boundedFix} \) then applies the algebra to that value recursively. In essence \( \text{boundedFix} \) 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 \( \text{EvalX} \). The definition uses a distinguished bottom value, \( \bot \), that represents a computation which does not finish within the given bound.

\[
\begin{align*}
\text{data } \bot_F a = \text{Bot} \\
\bot = \text{inject } \text{Bot} \\
\text{evalX } : (\text{Function } f, \bot_F \prec r, \text{Eval}_X f \prec r) \Rightarrow \\
\text{Fix}_M f \to \text{Fix}_M r \\
\text{evalX } e = \text{boundedFix } 1000 (\bot_{\Rightarrow} \bot) \text{ eval}_{\text{Mix}} [e]
\end{align*}
\]

5.4 Backwards compatibility

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

1. \( \text{eval}_X \) uses mixin instead of Mendler algebras.

2. \( \text{eval}_X \) 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 \( \text{mendlerToMixin} \) function automatically derives the required mixin algebra from the Mendler algebra.

\[
\begin{align*}
\text{mendlerToMixin } : \text{Algebra}_M f a \to \text{Mendler } (\text{Fix}_M g) f a \\
\text{mendlerToMixin } alg = alg
\end{align*}
\]

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

\[
\begin{align*}
\text{instance } \text{Eval } \text{Arith}_F f \Rightarrow \text{EvalX } f \text{ Arith}_F r \text{ where} \\
\text{eval}_{\text{Mix}} [\ ] e env = \\
\text{mendlerToMixin } \text{evalAlgebra } (\text{flip } [\ ] env) e
\end{align*}
\]

The algebras of binder-free features can be similarly adapted to build an algebra over a parameteric 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.

**Theorem** \( \text{soundnessX } :: \forall f \in \Gamma t. \Gamma \upharpoonright \text{env } f t \Rightarrow \forall e_1 : \text{Fix}_M f (\text{Maybe } (\text{Fix}_M f)) \\
\forall e_2 : \text{Fix}_M (\text{Fix}_M f) \text{ Nat}. \Gamma \vdash e_1 \equiv e_2 \Rightarrow \\
type e_1 = \text{Just } t \Rightarrow WTValue (\text{evalX } e_2 \text{ env}) t
\]

The proof of \( \text{soundnessX} \) features two substantial changes to the proof of \( \text{soundness} \) from Section 4.3.

6.1 Proofs over Parametric Church Encodings

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

In recursive invocations of \( \text{soundnessX} \), 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 \( \Gamma \) 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, can then analyze e1 and e2 in lockstep. Figure 5 shows the rules for determining equivalence of lambda expressions.

\[
\begin{align*}
(x, x') & \in \Gamma \\
\Gamma \vdash \text{var} x & \equiv \text{var} x' & \text{(EQV-VAR)} \\
\Gamma \vdash e_1 & \equiv e_1' & \text{(EQV-APP)} \\
\Gamma \vdash \text{app} e_1 & \equiv \text{app} e_1' e_2 & \text{(EQV-APP)} \\
\forall x' \cdot (x, x'), \Gamma \vdash f(x) & \equiv f'(x') & \text{(EQV-ABS)}
\end{align*}
\]

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 mixing-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, the bound has been reached and evalX returns ⊥. The proof of this case relies on adding to the WTValue judgment the WF-Bot rule stating that every type is inhabited by ⊥.

\[
\bot \vdash T \text{ (WF-Bot)}
\]

Hence, whenever evaluation returns ⊥, 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 e1 e2 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 evalX e1 env produces a well-formed closure Clo s1 env', we must then show that evaluating e2 under the (evalX e2 env): env' environment is also well-formed. However, e2 is not a subterm of App e1 e2, so the conventional induction hypothesis for subterms does not apply. However, because evalX e1 (evalX e2 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 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. 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:

\[
\Gamma \vdash n \equiv \hat{b} \text{ (EQV-NP)}
\]

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

Equiv-NP states that the same constructor \(C\) of \(f_{np}\), applied to equivalent subterms \(\hat{a}\) and \(\hat{b}\), 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.

\[\text{instance} (PAly N P f_{np}) \Rightarrow iPAly N P (\text{EQV-NP f_{np}})\]

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 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:

\[
\begin{align*}
type & \text{ soundness } e : tp ev = \\
\forall ev t tp (out t (x e)) & = Just t \\
& \Rightarrow \text{WTValue (ev out t' (x e)) env t})
\]

\[
\begin{align*}
\text{type soundness}_{alg rec} & \text{ rec} \rightarrow \\
(typeof \text{alg rec} & \text{ rec}) (\text{evalalg rec}) \rightarrow \text{soundness e rec rec}'
\end{align*}
\]

The hypothesis \(IH\) is used to relate calls of \(\text{rec}\) and \(\text{rec}'\) to applications of \(\text{evalalg}\) and \(\text{typeofalg}\).

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.

\[\text{instance} (PAly N P E) \Rightarrow iPAly N (IH \rightarrow P) E\]

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. 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
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 \( \text{nat} \) are natural numbers: \( \vdash x : \text{nat} \rightarrow x : \mathbb{N} \). The \( \text{Bool} \) feature introduces a new typing judgment, \( \text{WT-BOOL} \) for boolean values. Any language which includes both these features must have an instance of this inversion for \( \text{WT-BOOL} \). Our modular approach supports feature interactions by capturing them in type classes. A missing case, like for \( \text{WT-BOOL} \), 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 \( \text{WF} \) discussed in Section 3.3.1. These tactics help minimizing the number of interaction type class instances, which could otherwise 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) \( \text{F-algebras} \) 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 \( \text{F-algebras} \): instead, they use an isomorphic representation called \text{object algebras} [31]. Object algebras are a better fit for languages where records are the main structuring construct (such as \( \text{OO} \) languages). Our solution differs from previous approaches in the use of Mendler-style \( \text{F-algebras} \) instead of conventional \( \text{F-algebras} \) 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 \textit{monads} to structure semantics. Moggi [28] pioneered monads to model computation effects and structure denotation semantics. Liang et al. [25] introduced \textit{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