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. Unfortunately, the considerable effort involved in mechanizing proofs has prevented it from becoming standard practice. This cost can be amortized by reusing as much of existing mechanized formalizations as possible when building a new language or extending an existing one. One important challenge in achieving reuse is that the inductive definitions and proofs used in these formalizations are closed to extension. This forces language designers to cut and paste existing definitions and proofs in an ad-hoc manner and to expend considerable effort to patch up the results.

The key contribution of this paper is the development of an induction technique for extensible Church encodings using 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. This framework enables a more structured approach to the reuse of meta-theory formalizations through the composition of modular inductive definitions and proofs.

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.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

Keywords Modular Mechanized Meta-Theory, Extensible Church Encodings, Coq

1. Introduction

With their POPLMARK challenge, Aydemir et al. [3] 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 three person-year verified compiler project [22], which consists of eight 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 unnecessarily labor-intensive. Moreover, from a software/formalization management perspective a proliferation of copies is obviously a nightmare. Typical formalizations present two important challenges to providing reuse:

1. **Extensibility:** Conventional inductive definitions and proofs are *closed* to extension and cannot simply be imported and extended with new constructors and cases. This is a manifestation of the well-known *Expression Problem* (EP) [45].
2. **Modular reasoning:** Reasoning with modular definitions requires reasoning about partial definitions and composing partial proofs to build a complete proof. However, conventional induction principles which are the fundamental reasoning techniques in most theorem provers only work for 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, 42, 46] and expressive forms of *dependent types* [11, 34]. It is reasonable to wonder whether these language constructs can help to achieve better reuse. After all, there has been a lot of progress in addressing extensibility [14, 30, 32, 43] 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 and reasoning about extensible inductive datatypes. The framework is implemented as a Coq library which enables modular mechanized meta-theory by allowing language features to be defined as reusable components. Using MTC, language developers can leverage existing efforts to build new languages by developing new and interesting features and combining them with previously written components.

The solution to extensibility in MTC was partly inspired by the popular "Data types à la Carte" (DTC) [43] technique. However, DTC fundamentally relies on a type-level fixpoint definition for building modular data types, which cannot be encoded in Coq. MTC solves this problem by using (extensible) *Church encodings* of data types [30, 34, 35]. These encodings allow DTC-style modular data types to be defined 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 [44] 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 *inductively defined predicates*.

MTC's solution to modular reasoning uses a novel reinterpretation of the *universal property* of folds. Because MTC relies on

POPL'13, January 23–25, 2013, Rome, Italy.

folds, the proof methods used in the *initial algebra semantics of data types* [16, 27] offer an alternative to structural induction. With some care, universal properties can be exploited to adapt these techniques to modular Church encodings. In addition to enabling modular reasoning about extensible inductive datatypes, universal properties also overcome some theoretical issues related to Church encodings in the Calculus of (Inductive) Constructions [34, 35].

MTC also supports ubiquitous higher-order language features such as binders and general recursion. Binders are modeled with a *parametric HOAS* [8] representation (a first-order representation would be possible too). Because these features require general recursion, they cannot be defined inductively using folds. To support these non-inductive features MTC uses a variation of mixins [10]. 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 the utility of MTC, we present a case study modularizing several orthogonal features of a variant of mini-ML [9]. 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 defining and reasoning about extensible inductive datatypes. MTC is a Coq framework for building reusable components that implements this approach to modular mechanized meta-theory.

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 principles for 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 user'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, the paper adopts a terser syntax for its code fragments. For the computational parts, this syntax exactly coincides with Haskell syntax, while it is an extrapolation of Haskell syntax for 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 the DTC solution to the Expression Problem in Haskell, in particular its composition mechanisms for extensible inductive definitions. MTC differs from DTC in two important ways. Firstly, it uses Church encodings to avoid the termination issues of DTC's generally recursive definitions. Secondly, it uses Mendler-style folds 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*. 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 arithmetic expression language with literals and addition.

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

The explicitly recursive definition $Fix_{DTC}\ f$ 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.

$$\mathbf{type}\ Algebra\ f\ a = f\ a \rightarrow a$$

$$\begin{array}{l} fold_{DTC} :: Functor\ f \Rightarrow Algebra\ f\ a \rightarrow Fix_{DTC}\ f \rightarrow a \\ fold_{DTC}\ alg\ (In\ fa) = alg\ (fmap\ (fold_{DTC}\ alg)\ fa) \end{array}$$

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

$$\mathbf{data}\ Value = I\ Int \mid B\ Bool$$

$$\begin{array}{ll} eval_{Arith} :: Algebra\ Arith_F\ Value & \\ eval_{Arith}\ (Lit\ n) & = I\ n \\ eval_{Arith}\ (Add\ (I\ v_1)\ (I\ v_2)) & = I\ (v_1 + v_2) \end{array}$$

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:

$$\begin{array}{l} [\![\cdot]\!] :: Fix_{DTC}\ Arith_F \rightarrow Value \\ [\![\cdot]\!] = fold_{DTC}\ eval_{Arith} \end{array}$$

$$\begin{array}{l} > [\![(Add\ (Lit\ 1)\ (Lit\ 2))]\!] \\ 3 \end{array}$$

Unfortunately, DTC's two uses of general recursion are not permitted in Coq. Coq does not accept the type-level fixpoint combinator $Fix_{DTC}\ f$ because it is not strictly positive. Coq similarly disallows the $fold_{DTC}$ function because it is not structurally recursive.

2.2 Recursion-Free Church Encodings

MTC encodes data types and folds with Church encodings [6, 35], which are recursion-free. Church encodings represent (least) fixpoints and folds as follows:

$$\mathbf{type}\ Fix\ f = \forall a. Algebra\ f\ a \rightarrow a$$

$$\begin{array}{l} fold :: Algebra\ f\ a \rightarrow Fix\ f \rightarrow a \\ fold\ alg\ fa = fa\ alg \end{array}$$

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

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

Example Church encodings of $Arith_F$'s literals and addition are given by the lit and add functions:

$$
\begin{aligned}
&lit :: Nat \to Fix\ Arith_F \\
&lit\ n = \lambda alg \to alg\ (Lit\ n) \\
&add :: Fix\ Arith_F \to Fix\ Arith_F \to Fix\ Arith_F \\
&add\ e_1\ e_2 = \lambda alg \to alg\ (Add\ (fold\ alg\ e_1)\ (fold\ alg\ e_2))
\end{aligned}
$$

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 therefore 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:

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

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

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

However, an important difference with the big-step semantics above 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. Accordingly, 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. Furthermore, this approach can be quite brittle: in more complex object languages using folds and laziness can lead to subtle semantic issues [4].

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 [44] 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 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:

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

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

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

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, MTC regulates 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:

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

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

$$
\begin{aligned}
\mathbf{data}\ Exp &= Lit\ Nat \mid Add\ Exp\ Exp \\
&\mid If\ Exp\ Exp\ Exp \mid BLit\ Bool
\end{aligned}
$$

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

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

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

Algebras for composite functor are built from feature algebras:

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

Overall evaluation can then be defined as:

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

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 continues to use more specific classes to make a gentler progression for the reader.

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

$$
\begin{array}{l}
\textbf{class } f \prec: g \textbf{ where} \\
\quad \mathit{inj} \quad\;\; :: f\ a \rightarrow g\ a \\
\quad \mathit{prj} \quad\;\; :: g\ a \rightarrow \mathit{Maybe}\ (f\ a) \\
\quad \mathit{inj\_prj} :: \mathit{prj}\ ga = \mathit{Just}\ \mathit{fa} \rightarrow ga = \mathit{inj}\ \mathit{fa} \quad \text{-- law} \\
\quad \mathit{prj\_inj} :: \mathit{prj} \circ \mathit{inj} = \mathit{Just} \quad\quad\quad\quad\quad\quad\;\; \text{-- law} \\
\textbf{instance } (f \prec: g) \Rightarrow f \prec: (g \oplus h) \textbf{ where} \\
\quad \mathit{inj}\ \mathit{fa} \quad\quad\;\; = \mathit{Inl}\ (\mathit{inj}\ \mathit{fa}) \\
\quad \mathit{prj}\ (\mathit{Inl}\ ga)\ = \mathit{prj}\ ga \\
\quad \mathit{prj}\ (\mathit{Inr}\ ha) = \mathit{Nothing} \\
\textbf{instance } (f \prec: h) \Rightarrow f \prec: (g \oplus h) \textbf{ where} \\
\quad \mathit{inj}\ \mathit{fa} \quad\quad\;\; = \mathit{Inr}\ (\mathit{inj}\ \mathit{fa}) \\
\quad \mathit{prj}\ (\mathit{Inl}\ ga)\ = \mathit{Nothing} \\
\quad \mathit{prj}\ (\mathit{Inr}\ ha) = \mathit{prj}\ ha \\
\textbf{instance } f \prec: f \textbf{ where} \\
\quad \mathit{inj}\ \mathit{fa} = \mathit{fa} \\
\quad \mathit{prj}\ \mathit{fa} = \mathit{Just}\ \mathit{fa}
\end{array}
$$

Figure 1. Functor subtyping.

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 [42] perform a backtracking search for matching instances. Hence, 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.

$$
\begin{array}{l}
in_f :: f\ (\mathit{Fix}_M\ f) \rightarrow \mathit{Fix}_M\ f \\
in_f\ \mathit{fexp} = \lambda \mathit{alg} \rightarrow \mathit{alg}\ (\mathit{fold}_M\ \mathit{alg})\ \mathit{fexp}
\end{array}
$$

Smart constructors are built using in_f and inj as follows:

$$
\begin{array}{l}
\mathit{inject} :: (g \prec: f) \Rightarrow g\ (\mathit{Fix}_M\ f) \rightarrow \mathit{Fix}_M\ f \\
\mathit{inject}\ \mathit{gexp} = in_f\ (\mathit{inj}\ \mathit{gexp}) \\
\mathit{lit} :: (\mathit{Arith}_F \prec: f) \Rightarrow \mathit{Nat} \rightarrow \mathit{Fix}_M\ f \\
\mathit{lit}\ n = \mathit{inject}\ (\mathit{Lit}\ n) \\
\mathit{blit} :: (\mathit{Logic}_F \prec: f) \Rightarrow \mathit{Bool} \rightarrow \mathit{Fix}_M\ f \\
\mathit{blit}\ b = \mathit{inject}\ (\mathit{BLit}\ b) \\
\mathit{cond} :: (\mathit{Logic}_F \prec: f) \\
\quad\quad \Rightarrow \mathit{Fix}_M\ f \rightarrow \mathit{Fix}_M\ f \rightarrow \mathit{Fix}_M\ f \rightarrow \mathit{Fix}_M\ f \\
\mathit{cond}\ c\ e_1\ e_2 = \mathit{inject}\ (\mathit{If}\ c\ e_1\ e_2)
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{exp} :: \mathit{Fix}_M\ (\mathit{Arith}_F \oplus \mathit{Logic}_F) \\
\mathit{exp} = \mathit{cond}\ (\mathit{blit}\ \mathit{True})\ (\mathit{lit}\ 3)\ (\mathit{lit}\ 2) \\
> \mathit{eval}\ \mathit{exp} \\
3
\end{array}
$$

The out_f function exposes the toplevel functor again:

$$
\begin{array}{l}
\mathit{out}_f :: \mathit{Functor}\ f \Rightarrow \mathit{Fix}_M\ f \rightarrow f\ (\mathit{Fix}_M\ f) \\
\mathit{out}_f\ \mathit{exp} = \mathit{fold}_M\ (\lambda \mathit{rec}\ \mathit{fr} \rightarrow \mathit{fmap}\ (in_f \circ \mathit{rec})\ \mathit{fr})\ \mathit{exp}
\end{array}
$$

We can pattern match on particular features using prj and out_f:

$$
\begin{array}{l}
\mathit{project} :: (g \prec: f, \mathit{Functor}\ f) \Rightarrow \\
\quad \mathit{Fix}_M\ f \rightarrow \mathit{Maybe}\ (g\ (\mathit{Fix}_M\ f)) \\
\mathit{project}\ \mathit{exp} = \mathit{prj}\ (\mathit{out}_f\ \mathit{exp}) \\
\mathit{isLit} :: (\mathit{Arith}_F \prec: f, \mathit{Functor}\ f) \Rightarrow \mathit{Fix}_M\ f \rightarrow \mathit{Maybe}\ \mathit{Nat} \\
\mathit{isLit}\ \mathit{exp} = \textbf{case } \mathit{project}\ \mathit{exp} \textbf{ of} \\
\quad \mathit{Just}\ (\mathit{Lit}\ n) \rightarrow \mathit{Just}\ n \\
\quad \mathit{Nothing} \quad\quad\; \rightarrow \mathit{Nothing}
\end{array}
$$

2.6 Extensible Semantic Values

In addition to modular 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 sorts of extensional encodings as the expression language itself:

$$
\begin{array}{l}
\textbf{data } \mathit{NVal}_F\ a\ = I\ \mathit{Nat} \\
\textbf{data } \mathit{BVal}_F\ a\ = B\ \mathit{Bool} \\
\textbf{data } \mathit{Stuck}_F\ a = \mathit{Stuck} \\
\mathit{vi} :: (\mathit{NVal}_F \prec: r) \Rightarrow \mathit{Nat} \rightarrow \mathit{Fix}_M\ r \\
\mathit{vi}\ n = \mathit{inject}\ (I\ n) \\
\mathit{vb} :: (\mathit{BVal}_F \prec: r) \Rightarrow \mathit{Bool} \rightarrow \mathit{Fix}_M\ r \\
\mathit{vb}\ b = \mathit{inject}\ (B\ b) \\
\mathit{stuck} :: (\mathit{Stuck}_F \prec: r) \Rightarrow \mathit{Fix}_M\ r \\
\mathit{stuck} = \mathit{inject}\ \mathit{Stuck}
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{class } \mathit{Eval}\ f\ r \textbf{ where} \\
\quad \mathit{eval}_{alg} :: \mathit{Algebra}_M\ f\ (\mathit{Fix}_M\ r)
\end{array}
$$

Projection is now essential for pattern matching on values:

$$
\begin{array}{l}
\textbf{instance } (\mathit{Stuck}_F \prec: r, \mathit{NVal}_F \prec: r, \mathit{Functor}\ r) \Rightarrow \\
\quad \mathit{Eval}\ \mathit{Arith}_F\ r \textbf{ where} \\
\quad\quad \mathit{eval}_{alg}\ [\![\cdot]\!]\ (\mathit{Lit}\ n) \quad\quad = \mathit{vi}\ n \\
\quad\quad \mathit{eval}_{alg}\ [\![\cdot]\!]\ (\mathit{Add}\ e_1\ e_2) = \\
\quad\quad\quad \textbf{case } (\mathit{project}\ [\![e_1]\!], \mathit{project}\ [\![e_2]\!]) \textbf{ of} \\
\quad\quad\quad\quad (\mathit{Just}\ (I\ n_1), (\mathit{Just}\ (I\ n_2))) \rightarrow \mathit{vi}\ (n_1 + n_2) \\
\quad\quad\quad\quad \_ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\;\; \rightarrow \mathit{stuck}
\end{array}
$$

This concludes MTC's support for extensible inductive data types and functions. To cater to meta-theory, MTC must also support 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.

$$
\begin{array}{l}
A_{ind} :: \forall P :: (\mathit{Arith} \rightarrow \mathit{Prop}). \\
\quad\quad \forall H_l :: (\forall n. P\ (\mathit{Lit}\ n)). \\
\quad\quad \forall H_a :: (\forall a\ b. P\ a \rightarrow P\ b \rightarrow P\ (\mathit{Add}\ a\ b)). \\
\quad\quad \forall a. P\ a \\
A_{ind}\ P\ H_l\ H_a\ e = \\
\quad \textbf{case } e \textbf{ of } \mathit{Lit}\ n \rightarrow H_l\ n \\
\quad\quad \mathit{Add}\ x\ y \quad\quad \rightarrow H_a\ a\ b\ (A_{ind}\ P\ H_l\ H_a\ x) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (A_{ind}\ P\ H_l\ H_a\ y)
\end{array}
$$

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, so 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.

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

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 reasoning about semantic functions requires a new approach that does not suffer from this restriction.

3.2 Type Dependency with Dependent Products

Hutton's first aspect of *fold*s is that they become substantially more expressive with the help of tuples. The dependent products in Coq take this observation one step further. While an f-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:

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

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.

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

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.

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

This constraint is encoded in the typeclass for proof algebras, $PAlg$. It is easy to verify that A^{2}_{ind} satisfies this property. Other proof algebras over $Arith_F$ can be defined by instantiating A^{2}_{ind} with appropriate cases for H_l and H_a. 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 proof 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 $Fix_M\ f$. Thus, the only hard well-formedness requirement for a Church-encoded term e is that it satisfies the right-to-left implication of the universal property.

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

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:

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

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, these properties provide a handle on reasoning about these terms.

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

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

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 prove the key theorem for building 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 can conclude 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:

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

□

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. These proofs should be independent from one another, so that they can be reused for different combinations of features.

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 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:

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

Consider building a proof of

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

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

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

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. To proceed, we need the additional property that this f-algebra delegates to the $Arith_F$-algebra as expected:

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

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.

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

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

Figure 2. WF_Eval instances.

MTC provides the three instances of this class in Figure 2, one for each instance of $\prec:$, allowing Coq to automatically build a proof of well-formedness for every 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.

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 Inductive Predicates

Many proofs appeal to rules which define a predicate for an important property. In Coq these predicates 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.

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

When dealing with a predicate over extensible inductive data types, the set of rules defining the predicate must be extensible as well. Extensibility of these rules 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.

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

This index 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:

$$
\begin{array}{l}
\mathbf{type}\ iAlg\ i\ (f :: (i \to Prop) \to (i \to Prop))\ a \\
\quad = \forall x :: i. f\ a\ x \to a\ x \\
\mathbf{type}\ iFix\ i\ (f :: (i \to Prop) \to (i \to Prop))\ (x :: i) \\
\quad = \forall a :: i \to Prop. iAlg\ f\ a \to a\ x\ ...
\end{array}
$$

As these indexed variants are meant to construct logical relations, their parameters range over $Prop$ instead of Set. Fortunately,

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** $inj :: f\ a \to g\ a$ $prj :: g\ a \to Maybe\ (f\ a)$ $inj\_prj :: prj\ ga = Just\ fa \to$ $ga = inj\ fa$ $prj\_inj :: prj \circ inj = Just$	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 $(f \prec: g, FAlg\ n\ t\ a\ f, FAlg\ n\ t\ a\ g) \Rightarrow$ $WF\_FAlg\ n\ t\ a\ f\ g$ **where** $wf\_algebra :: \forall rec\ (fa :: f\ t).$ $f\_algebra\ rec\ (inj\ fa) =$ $f\_algebra\ rec\ fa$	Algebra Delegation Inferred
class $(Functor\ f, Functor\ g, f \prec: g) \Rightarrow$ $PAlg\ name\ f\ g\ a$ **where** $p\_algebra :: Algebra\ f\ a$ $proj\_eq :: \forall e.\pi_1\ (p\_algebra\ e) =$ $in_f\ (inj\ (fmap\ \pi_1\ e))$	**Proof Algebras** Supplied by the User

Figure 3. Type classes provided by MTC

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:

$$\textbf{data}\ TNat_F\ t\ = TNat$$
$$\textbf{data}\ TBool_F\ t = TBool$$

For this language soundness is formulated as:

Theorem $soundness ::$
$\forall e\ t\ env, typeof\ e = Just\ t \to WTValue\ (eval\ e\ env)\ t$

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 every case analysis to be captured in a sublemma. Because the superfunctor is abstract, the cases are not known locally and must 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:

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

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. 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 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.

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

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 components are 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:

[2] Actually, as proofs are opaque, we cannot tell if they are equal.

Theorem $WTV_{\pi_1,v}\ (i :: WTValue\ v\ t) =$
$\forall v'.\pi_1\ v = \pi_1\ v' \to WTValue\ v'\ t$

Theorem $WTV_{\pi_1,t}\ (i :: WTValue\ v\ t) =$
$\forall t'.\pi_1\ t' = \pi_1\ t' \to WTValue\ v\ t'$

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) [8] 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 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 usual type parameter r for recursive occurrences.

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

As before, smart constructors build extensible expressions:

$$var :: (Lambda_F\ v \prec: f) \Rightarrow v \to Fix_M\ f$$
$$var\ v = inject\ (Var\ v)$$
$$app :: (Lambda_F\ v \prec: f) \Rightarrow Fix_M\ f \to Fix_M\ f \to Fix_M\ f$$
$$app\ e_1\ e_2 = inject\ (App\ e_1\ e_2)$$
$$lam :: (Lambda_F\ v \prec: f) \Rightarrow (v \to Fix_M\ f) \to Fix_M\ f$$
$$lam\ f = inject\ (Lam\ f)$$

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:

$$\mathbf{data}\ Value =$$
$$Stuck \mid I\ Nat \mid B\ Bool \mid Clos\ (Value \to Value)$$

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.

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

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

$$\mathbf{data}\ Closure_F\ f\ a = Clos\ (Fix_M\ f)\ (Env\ a)$$
$$closure :: (Closure_F\ f \prec: r) \Rightarrow$$
$$Fix_M\ f \to Env\ (Fix_M\ r) \to Fix_M\ r$$
$$closure\ mf\ e = inject\ (Clos\ mf\ e)$$

A first attempt at defining evaluation is:

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

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.

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

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:

$$eval_{Lambda} :: (Closure_F\ e \prec: v, Stuck_F \prec: v) \Rightarrow$$
$$Mixin\ (Fix_M\ e)\ (Lambda_F\ Nat)$$
$$(Env\ (Fix_M\ v) \to Fix_M\ v)$$

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

$$\mathbf{class}\ Eval_X\ f\ g\ r\ \mathbf{where}$$
$$eval_{xalg} :: Mixin\ (Fix_M\ f)\ g\ (Env\ (Fix_M\ r) \to Fix_M\ r)$$
$$\mathbf{instance}\ (Stuck_F \prec: r, Closure_F\ f \prec: r, Functor\ r) \Rightarrow$$
$$Eval_X\ f\ (Lambda_F\ Nat)\ r\ \mathbf{where}$$
$$eval_{xalg} = eval_{Lambda}$$

Although the code of $eval_{Lambda}$ still appears generally recursive, it is actually *not* because the recursive calls are abstracted as a parameter (like with Mendler algebras). Accordingly, $eval_{Lambda}$ 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 [10]. 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.

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

The argument e is a Mendler-encoded expression of type $\mathit{Fix}_M\ f$. *boundedFix* first uses out_f to unfold the expression into a value of type $f\ (\mathit{Fix}_M\ f)$ and 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, $\perp$, that represents a computation which does not finish within the given bound.

$$\begin{aligned}
&\mathbf{data}\ \perp_F\ \ a = \mathit{Bot} \\
&\perp = \mathit{inject}\ \mathit{Bot} \\
&\mathit{eval}_X :: (\mathit{Functor}\ f, \perp_F \prec: r, \mathit{Eval}_X\ f\ f\ r) \Rightarrow \\
&\quad \mathit{Nat} \to \mathit{Fix}_M\ f \to \mathit{Env} \to \mathit{Fix}_M\ r \\
&\mathit{eval}_X\ n\ e\ \mathit{env} = \mathit{boundedFix}\ n\ (\backslash_ \to \perp)\ \mathit{eval}_{xalg}\ e\ \mathit{env}
\end{aligned}$$

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 algebras instead of Mendler algebras.
2. eval_X now expects algebras over a parameterized functor.

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:

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

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.

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

The proof of $\mathit{soundness}_X$ features two substantial changes to the proof of *soundness* from Section 4.3.

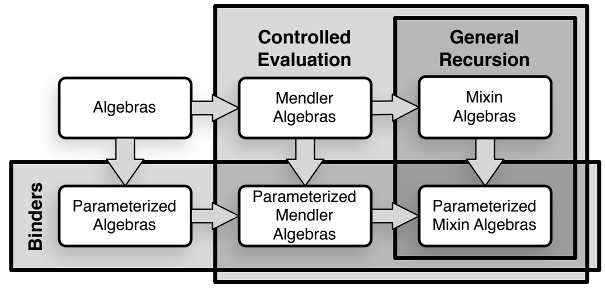

Figure 4. Hierarchy of Algebra Adaptation

6.1 Proofs over Parametric Church Encodings

The statement of $\mathit{soundness}_X$ uses two instances of the same PHOAS expression $e :: \forall v.\mathit{Fix}_M\ (f\ \ v)$. The first, e_1, instantiates v with the appropriate type for the typing algebra, while e_2 instantiates v for the evaluation algebra.

In recursive applications of $\mathit{soundness}_X$, the connection between e_1 and e_2 is no longer apparent. As they have different types, Coq considers them to be distinct, so case analysis on one does not convey information about the other. Chlipala [8] shows how the connection can be retained with the help of an auxiliary equivalence relation $\Gamma \vdash e_1 \equiv e_2$, which uses the environment Γ to keep track of the current variable bindings. The top-level application, 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 applications of $\mathit{soundness}_X$ can then analyze e_1 and e_2 in lockstep. Figure 5 shows the rules for determining equivalence of lambda expressions.

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

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

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 $\forall n\ \ e, (\forall e', P(f\ \ n\ \ e')) \to \forall e, P(f\ (n+1)\ e)$ hold, $\forall n\ e, P(f\ n\ e)$ also holds.

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

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

Hence, whenever evaluation returns $\perp$, 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: $\mathit{IH} :: \forall n\ e, (\forall e', P(f\ n\ e')) \to P(f\ (n+1)\ e)$. The $\mathit{App}\ \ e_1\ \ e_2$ 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. 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, existing proof algebras for those features need to be adapted. To cater to the four possible proof signatures of soundness (one for each definition of $[\![\cdot]\!]$), 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:

$$\frac{\Gamma \vdash \bar{a} \equiv \bar{b}}{\Gamma \vdash inject(C\ \bar{a}) \equiv inject(C\ \bar{b})} \qquad (\textrm{EQV-NP})$$

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

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

This instance requires a small concession: proofs over f_{np} have to be stated in terms of two expressions with distinct superfunctors f and f' rather than two occurrences of the same expression. Induction over these two expressions requires a variant of $PAlg$ 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:

$$\begin{array}{l}
\mathbf{type}\ soundness\ e\ tp\ ev = \\
\quad \forall env\ t.tp\ (out_f\ (\pi_1\ e)) = Just\ t \rightarrow \\
\quad\quad WTValue\ (ev\ (out_t'\ (\pi_1\ e))\ env)\ t)
\end{array}$$

$$\begin{array}{l}
\mathbf{type}\ soundness_{alg}\ rec_t\ rec_e \\
\quad (typeof_{alg} :: Mixin\ (Fix_M\ f)\ f\ (Maybe\ (Fix_M\ t)) \\
\quad (eval_{alg} :: Mixin\ (Fix_M\ f)\ f\ (Env\ (Fix_M\ r) \rightarrow Fix_M\ r)) \\
\quad (e :: Fix_M\ f)\ (e_UP' :: UP\ e) = \\
\quad\quad \forall IHc :: (\forall e'. \\
\quad\quad\quad soundness\ e'\ (typeof_{alg}\ rec_t)\ (eval_{alg}\ rec_e) \rightarrow \\
\quad\quad\quad soundness\ e'\ rec_t\ rec_e). \\
\quad\quad soundness\ e\ (typeof_{alg}\ rec_t)\ (eval_{alg}\ rec_e)
\end{array}$$

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

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 a proof algebra with this signature to one that includes the Induction Hypothesis generated by induction on the bound of $boundedFix$.

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

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 [9] 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 feature that provides that set of definitions.

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.

e ::= ℕ | e + e	*Arith*
| 𝔹 | **if** e **then** e **else** e	*Bool*
| **case** e **of** { **z** ⇒ e ; **S** n ⇒ e}	*NatCase*
| **lam** x : T.e | e e | x	*Lambda*
| **fix** x : T.e	*Recursion*

V ::= ℕ	*Arith*
| 𝔹	*Bool*
| **closure** e $\overline{\texttt{V}}$	*Lambda*

T ::= **nat**	*Arith*
| bool	*Bool*
| T → T	*Lambda*

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

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* [5], well-known from modular component-based frameworks. In essence, a feature interaction is functionality (e.g., a function or 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: $\vdash\ \texttt{x}\ :\ \mathbf{nat} \rightarrow \texttt{x}\ ::\ \mathbb{N}$. 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. These tactics help minimize the number of interaction type class instances, which could otherwise easily grow exponentially in the number of features.

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

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* [11], also use Church encodings to model inductive data types [35]. However, the inductive principles to reason about those encodings had to be axiomatized, which 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. MTC returns to Church encodings to allow extensibility, but does not use standard, closed induction principles. It instead uses a reasoning framework based on universal properties which allow modular reasoning 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 [14, 43]. However the type-level fixpoints that are central to DTC cannot be used in Coq because of their use of general recursion. To avoid general recursion, MTC encodes *least-fixpoints* with Church encodings [6, 35]. Church encodings have inspired other solutions to the expression problem (especially in object-oriented languages) [30–32]. 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). MTC differs from previous approaches by using 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, MTC also deals with modular reasoning and extensibile inductively defined predicates.

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. The Ott tool [41] allows users to write definitions and theorem statements in an ASCII format designed to mirror pen-and-paper formalizations. These are then automatically translated to definitions in either LaTeX or a theorem prover and proofs and functions are then written using the generated definitions.

Both Boite [7] 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 which are no longer available and have users 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. [13] applied product-line techniques to 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.

Concurrently with our development of MTC, Schwaab et al. have been working on modularizing meta-theory in Agda [40]. While MTC uses Church encodings to encode extensible datatypes, their approach achieves extensibility by using universes which can be lifted to the type level. Encodings and their associated proofs can be modified to derive new languages.

Transparency One long-standing criticism of mechanized metatheory has been that it interferes with adequacy, i.e. convincing users that the proven theorem is in fact the desired one [39]. Certainly the use of PHOAS can complicate the transparency of mechanized definitions. The $soundness_X$ theorem, for example, uses a more complicated statement than the pen-and-paper version because PHOAS requires induction over the equivalence relation. Modular inductive datatypes have the potential for exacerbating transparency concerns, as the encodings are distributed over different components. Combining a higher-level notation provided by a tool like Ott with the composition mechanisms of MTC is an interesting direction for future work. Such a higher-level notation could help with transparency; while MTC's composition mechanisms could help with generating modular code for Ott specifications.

Binding To minimize the work involved in modeling binders, MTC provides reusable binder components. The problem of modeling binders has previously received a lot of attention. Some 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 [8] avoids most infrastructure definitions. While we have developed PHOAS-based binders in MTC, it supports first-order representations as well.

Semantics and Interpreters While the majority of semantics formalization approaches use inductively defined predicates, we propose an approach based on interpreters. Of course, MTC supports standard approaches as well.

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 from using 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 have some modular proof techniques for reasoning about monads have been introduced [15, 33]. While these are good starting points, it remains to be seen whether these techniques are sufficient to reason about suitably generalized modular statements like soundness.

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 inductively defined relations. The only similar work in this direction, developed concurrently to our own, is that of Danielsson [12]. 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 language components. Furthermore, MTC supports modular reasoning about these components. Our approach enables reuse of modular inductive definitions and proofs that deal with standard language constructs, allowing language designers to 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 currently considering 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.

Acknowledgements We would like to especially thank William Cook for his help in structuring the presentation of this work. Further thanks to Don Batory, Twan van Laarhoven and the reviewers for their comments and suggestions. This work was supported by the National Science Foundation under Grant CCF 0724979.

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

[4] 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.

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

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

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

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

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

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

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

[12] N. A. Danielsson. Operational semantics using the partiality monad. In *ICFP'12*, 2012.

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

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

[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] Robert Pollack. How to believe a machine-checked proof. In *Twenty Five Years of Constructive Type Theory*, 1998.

[40] Christopher Schwaab and Jeremy G. Siek. Modular type-safety proofs using dependant types. *CoRR*, abs/1208.0535, 2012.

[41] Peter Sewell et al. Ott: effective tool support for the working semanticist. In *ICFP '07*, 2007.

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

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

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

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

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