Modular Monadic Meta-Theory

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

Steven Keuchel Tom Schrijvers
Ghent University
{steven.keuchel,tom.schrijvers}@ugent.be

Bruno C. d. S. Oliveira
The University of Hong Kong
bruno@cs.hku.hk

Abstract

This paper presents 3MT, a framework for *modular* mechanized meta-theory of languages with *effects*. Using 3MT, individual language features and their corresponding definitions – *semantic functions*, *theorem statements* and *proofs* – can be built separately and then reused to create different languages with fully mechanized meta-theory. 3MT combines *modular datatypes* and *monads* to define denotational semantics with effects on a per-feature basis, without fixing the particular set of effects or language constructs.

One well-established problem with *type soundness* proofs for denotational semantics is that they are notoriously brittle with respect to the addition of new effects. The statement of type soundness for a language depends intimately on the effects it uses, making it particularly challenging to achieve modularity. 3MT solves this long-standing problem by splitting these theorems into two separate and reusable parts: a *feature theorem* that captures the well-typing of denotations produced by the semantic function of an invidual feature with respect to only the effects used, and an *effect theorem* that adapts well-typings of denotations to a fixed superset of effects. The proof of type soundness for a particular language simply combines these theorems for its features and the combination of their effects. To establish both theorems, 3MT uses two key reasoning techniques: *modular induction* and *algebraic laws* about effects. Several effectful language features, including references and errors, illustrate the capabilities of 3MT. A case study reuses these features to build fully mechanized definitions and proofs for 28 languages, including several versions of mini-ML with effects.

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

General Terms Languages

Keywords mechanized meta-theory; modularity; monads; side-effects

1. Introduction

Theorem provers are actively used to mechanically verify large-scale formalizations of critical components, including programming language meta-theory [1], compilers [25], large mathematical proofs [15] and operating system kernels [23]. Due to their scale and complexity, these developments can be quite time consuming, often demanding multiple man-years of effort.

It is reasonable to expect that variations can simply extend and reuse the original development in order to leverage the large investment of resources in these formalizations. This is unfortunately often not the case, as even small extensions can require significant additional effort. Adding a new language feature to a programming language formalization or compiler, for example, involves significant redesigns that have a cross-cutting impact on nearly all definitions and proofs in the formalization. This leads to a copy-paste-patch approach to reuse with several modified copies of the original development, making it difficult to compose new features and ultimately leading to a maintenance nightmare. Dissatisfied with this situation, several researchers [1, 15, 43, 44] have called for better ways to modularize mechanical formalizations.

This work extends the current state-of-the-art in modular mechanizations by solving a well-known and long-standing open problem with denotational semantics: type soundness proofs are notoriously brittle with respect to the addition of new effects. This is an important problem because effects are pervasive in programming language formalizations: in addition to extensions to syntax and semantics, new features usually introduce new effects to the denotations. Without a more robust formulation of type soundness, the addition of new effects requires cross-cutting changes to type soundness theorem statements and proofs.

Initially the semantics themselves were also brittle with respect to effects [24, 32], but *monads* [31, 50] have been found to provide the necessary robustness to denotations. Yet as far as we know, the brittleness of (denotational) type soundness proofs has remained an open problem since it was raised by Wright and Felleisen [53] to motivate their own type-soundness approach. The framework we present here, *modular monadic meta-theory* (3MT), is the first in 20 years to provide a substantial solution. Using 3MT, we develop a novel approach to proving type soundness for monadic denotational semantics in a way that is modular in the set of effects used. Proofs for individual features do not depend on effects they do not use and hence are robust to extension.

The solution builds on Meta-Theory à la Carte (MTC) [9], a Coq framework for the mechanization of formal programming language meta-theory that supports modular extension of existing definitions. With MTC it is possible to develop meta-theory which is modular in two dimensions: *language features* on the one hand and *functions* and *proofs* over these features on the other hand. MTC adapts ideas from existing programming language solutions [34, 46] to the *expression problem* [51] for functions and features, and adds *modular induction* for proofs.

3MT adds a third modularity dimension to MTC: modular addition of new *effects*. 3MT enables the separate definition of features with effectful semantic functions and proofs over these functions, and reuse of these features in formalizations of multiple languages.

ICFP '13, September 25–27, 2013, Boston, MA, USA.

http://dx.doi.org/10.1145/2500365.2500587

To make denotations robust with respect to effects, 3MT uses the established solution, *monads*. In Coq, *type classes* [52] enable semantic function definitions that are constrained, yet polymorphic in the monad. This allows the inclusion of a feature in any language which supports a superset of its effects. When a language is composed from different effectful features, *monad transformers* [28] are used to instantiate the denotation's monad with all the effects required by the modular components.

To solve the key challenge of modularizing and reusing theorems and proofs of type soundness, we split the classic type soundness theorems into three parts:

1. Reusable *feature theorems* capture the essence of type soundness for an individual feature. They depend only on that feature's syntax, typing relation, semantic function and the effects used therein. At the same time, they abstract over the syntax, semantics and effects of other features. This means that the addition of new features with other types of effects *does not affect* the existing feature theorem proofs.
   To achieve the abstraction over other effects, a feature uses a constrained polymorphic monad. As a consequence, it only establishes the well-typing of the resulting denotations with respect to the effects declared in the constraints.
2. Reusable *effect theorems* fix the monad of denotations and consequently the set of effects. They take well-typing proofs of monadic denotations expressed in terms of a constrained polymorphic monad and which mention only a subset of effects, and turn them into well-typings with respect to a fixed monad and all the effects it provides.
   Effect theorems reason fully at the level of denotations and abstract over the details of language features like syntax and semantic functions.
3. Finally, *language theorems* establish type soundness for a particular language. They require no more effort than to instantiate the set of features and the set of effects (i.e., the monad), thus tying together the respective feature and effect theorems into an overall proof.

To establish the first two theorems, 3MT relies on modular induction and algebraic laws about effects. As far as we know, it applies the most comprehensive set of such laws to date, as each effect utilized by a feature needs to be backed up by laws and interactions between different effects must also be governed by laws. These laws are crucial for modular reasoning in the presence of effects.

In summary, the specific contributions of this work are:

- A reusable framework, 3MT, for mechanized meta-theory of languages with effects. This framework includes a mechanized library for monads, monad transformers and corresponding algebraic laws in Coq. Besides several laws for specific types of effects, the library also includes laws for the interactions between different types of effects.
- A new *modular* proof method for type-soundness proofs of denotational semantics.
- A case study of a family of fully mechanized languages, including a mini-ML variant with errors and references. The case study comprises 28 languages, 8 different effect theorems and 5 features with their feature theorems.

3MT is implemented in the Coq proof assistant and the code is available at `http://www.cs.utexas.edu/~bendy/3MT`.

Code and Notational Conventions While all the code underlying this paper has been developed in Coq, the paper adopts a terser syntax for its many code fragments. For the computational parts, this syntax exactly coincides with Haskell syntax, while it is an extrapolation of Haskell syntax style for propositions and proof concepts. Following MTC, the Coq code requires the impredicative-set option due to the use of Church encodings.

2. Background: Meta-Theory à la Carte

This section summarizes the necessary parts of the *Meta-Theory à la Carte* (MTC) approach to modular datatypes in Coq. For the full details of MTC, we refer the reader to the original paper [9].

2.1 Mendler Church Encodings and Folds for Semantics

MTC encodes data types and folds with a variant of Church encodings [5, 36] based on Mendler folds [47]. The advantage of Mendler folds is that recursive calls are explicit, allowing the user to precisely control the evaluation order. The Mendler-Church encodings represent (least) fixpoints and folds as follows:

$$\mathbf{type}\ \mathit{Algebra}_M\ f\ a = \forall r.(r \to a) \to f\ r \to a$$
$$\mathbf{type}\ \mathit{Fix}_M\ f = \forall a.\mathit{Algebra}_M\ f\ a \to a$$
$$\mathit{fold}_M :: \mathit{Algebra}_M\ f\ a \to \mathit{Fix}_M\ f \to a$$
$$\mathit{fold}_M\ \mathit{alg}\ \mathit{fa} = \mathit{fa}\ \mathit{alg}$$

Mendler algebras ($\mathit{Algebra}_M\ f\ a$) use a function argument of type $(r \to a)$ for their recursive calls. To enforce structurally recursive calls, arguments which appear at recursive positions have a polymorphic type r. Using this polymorphic type prevents case analysis, or any type of inspection, on those arguments. Mendler-Church encodings ($\mathit{Fix}_M\ f$) are functions of type $\forall a.\mathit{Algebra}_M\ f\ a \to a$. Mendler folds are defined by directly applying a Church encoded value fa to a Mendler algebra alg. All these definitions are non-recursive and can thus be expressed in Coq.

Example As a simple example, consider a language for boolean expressions supporting boolean literals and conditionals:

$$\mathbf{data}\ \mathit{Logic}_F\ e = \mathit{BLit}\ \mathit{Bool} \mid \mathit{If}\ e\ e\ e$$
$$\mathbf{type}\ \mathit{Value} = \mathit{Bool}$$

The evaluation algebra for this language is defined as follows:

$$\mathit{ifAlg} :: \mathit{Algebra}_M\ \mathit{Logic}_F\ \mathit{Value}$$
$$\mathit{ifAlg}\ [\![\cdot]\!]\ (\mathit{BLit}\ b) \quad = b$$
$$\mathit{ifAlg}\ [\![\cdot]\!]\ (\mathit{If}\ e_1\ e_2\ e_3) = \mathbf{if}\ [\![e_1]\!]\ \mathbf{then}\ [\![e_2]\!]\ \mathbf{else}\ [\![e_3]\!]$$

Unlike conventional Church encodings and folds, the recursive calls ($[\![\cdot]\!]$) are explicit and indicate the evaluation order.
The evaluation function simply folds the ifAlg algebra:

$$\mathit{eval} :: \mathit{Fix}_M\ \mathit{Logic}_F \to \mathit{Value}$$
$$\mathit{eval} = \mathit{fold}_M\ \mathit{ifAlg}$$

2.2 Modular Composition of Features

MTC adapts the *Data Types à la Carte* (DTC) [46] approach for composing f-algebras to Mendler algebras.

Modular Functors Because feature syntax is defined by means of functors, such as Logic_F, it can easily be composed with functor composition:

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

The syntax of a language of both conditional and simple arithmetic expressions, for example, is $\mathit{Fix}\ (\mathit{Arith}_F \oplus \mathit{Logic}_F)$ where

$$\mathbf{data}\ \mathit{Arith}_F\ e = \mathit{Lit}\ \mathit{Int} \mid \mathit{Add}\ e\ e$$

Feature semantics are expressed as Mendler algebras and can be composed in a similar way.

Type Classes Unlike DTC, MTC defines a number of type classes with laws in order to support proofs. These classes and laws are summarized in the table in Figure 1. The second column notes whether the base instances of a particular class are provided by the user or are automatically inferred with a default instance. Importantly, instances of all these classes for feature compositions (using $\oplus$) are built automatically.

Class Definition	Description
class $\mathit{Functor}\ f$ **where** $\mathit{fmap} :: (a \to b) \to (f\ a \to f\ b)$ $\mathit{fmap\_id} :: \mathit{fmap}\ \mathit{id} \equiv \mathit{id}$ $\mathit{fmap\_fusion} :: \forall g\ h.$ $\mathit{fmap}\ h \circ \mathit{fmap}\ g \equiv \mathit{fmap}\ (h \circ g)$	**Functors** Supplied by the user
class $f \prec: g$ **where** $\mathit{inj} :: f\ a \to g\ a$ $\mathit{prj} :: g\ a \to \mathit{Maybe}\ (f\ a)$ $\mathit{inj\_prj} :: \mathit{prj}\ \mathit{ga} \equiv \mathit{Just}\ \mathit{fa} \to$ $\mathit{ga} \equiv \mathit{inj}\ \mathit{fa}$ $\mathit{prj\_inj} :: \mathit{prj} \circ \mathit{inj} \equiv \mathit{Just}$	Functor Subtyping Inferred
class $(\mathit{Functor}\ f, \mathit{Functor}\ g, f \prec: g) \Rightarrow$ $\mathit{WF\_Functor}\ f\ g$ **where** $\mathit{wf\_functor} :: \forall a\ b\ (h :: a \to b).$ $\mathit{fmap}\ h \circ \mathit{inj} \equiv \mathit{inj} \circ \mathit{fmap}\ h$	Functor Delegation Inferred
class $(\mathit{Functor}\ h, f \prec: h, g \prec: h) \Rightarrow$ $\mathit{DistinctSubFunctor}\ f\ g\ h$ **where** $\mathit{inj\_discriminate} :: \forall a\ (\mathit{fe} :: f\ a)$ $(\mathit{ge} :: g\ a).\mathit{inj}\ \mathit{fe} \not\equiv \mathit{inj}\ \mathit{ge}$	Functor Discrimination Inferred
class $\mathit{FAlg}\ \mathit{name}\ t\ a\ f$ **where** $\mathit{f\_algebra} : \mathit{Mixin}\ t\ f\ a$	**Function Algebras** Supplied by the user
class $(f \prec: g, \mathit{FAlg}\ n\ t\ a\ f, \mathit{FAlg}\ n\ t\ a\ g) \Rightarrow$ $\mathit{WF\_FAlg}\ n\ t\ a\ f\ g$ **where** $\mathit{wf\_algebra} :: \forall \mathit{rec}\ (\mathit{fa} :: f\ t).$ $\mathit{f\_algebra}\ \mathit{rec}\ (\mathit{inj}\ \mathit{fa}) \equiv$ $\mathit{f\_algebra}\ \mathit{rec}\ \mathit{fa}$	Algebra Delegation Inferred
class $(\mathit{Functor}\ f, \mathit{Functor}\ g, f \prec: g) \Rightarrow$ $\mathit{PAlg}\ \mathit{name}\ f\ g\ a$ **where** $\mathit{p\_algebra} :: \mathit{Algebra}\ f\ a$ $\mathit{proj\_eq} :: \forall e.\pi_1\ (\mathit{p\_algebra}\ e) \equiv$ $\mathit{in}_f\ (\mathit{inj}\ (\mathit{fmap}\ \pi_1\ e))$	**Proof Algebras** Supplied by the User

Figure 1. Type classes provided by 3MT

The $\mathit{Functor}$ class provides the *fmap* method and is an adaptation of the corresponding type class in Haskell. In contrast with the Haskell version, the two functor laws are part of the definition. The class $\prec:$ represents a subtyping relation between two functors f and g. This class is an adaptation of the corresponding class in DTC and it includes two additional laws which govern the behavior of functor projection and injection ($\mathit{inj\_prj}$ and $\mathit{prj\_inj}$). The class $\mathit{WF\_Functor}$ ensures that *fmap* distributes through injection, and the class $\mathit{DistinctSubFunctor}$ ensures that injections from two different subfunctors are distinct. Unlike DTC, MTC defines a single generic Coq type class, FAlg, for the definition of semantic algebras. FAlg is indexed by the name of the semantic function (name). Note that the type Mixin:

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

is a slight generalization of Mendler algebras, which is useful for defining non-inductive language features such as general recursion or higher-order binders. The type class $\mathit{WF\_FAlg}$ provides a well-formedness condition for every composite algebra. Finally, the type class PAlg provides the definitions for proof algebras.

2.3 Modular Proofs

The main novelty of MTC is its modular approach to inductive proofs. Regular structural induction is not available for Church encodings, so MTC adapts the proof methods used in the *initial algebra semantics of data types* [14, 29] – in particular *universal properties* – to support modular inductive proofs over Church encodings. Proofs are written in the same modular style as functions, using proof algebras (class PAlg in Figure 1). These algebras are folded over the terms and can be modularly combined. Unlike function algebras, proof algebras are subject to an additional constraint which ensures the validity of the proofs ($\mathit{proj\_eq}$).

Sublemmas Each feature builds extensible datatypes by abstracting them over a super-functor. Because this super-functor is abstract, the complete set of cases needed by a proof algebra is unknown within a feature. To perform induction, a feature must therefore dispatch proofs to an abstract proof algebra over this super-functor. The components of this proof algebra are built in a distributed fashion among individual features. These components can then be composed to build a complete proof algebra for a concrete composition of functors.

As an example, consider the lemma that the type equality function eqType is sound:

$$\forall t_1\ t_2.\mathit{eqType}\ t_1\ t_2 \equiv \mathit{true} \to t_1 \equiv t_2 \qquad (\mathit{EqP})$$

This property can be captured in a proof algebra:

$$\mathit{PAlg}\ \mathit{EqF}_{\mathit{name}}\ f\ f\ (\exists\ e : \mathit{EqP}\ e)$$

A feature can build a proof of EqP for a specific type t by folding this proof algebra over t. Features also provide specific instances of this proof algebra for the types they introduce:

$$\mathit{PAlg}\ \mathit{EqF}_{\mathit{name}}\ \mathit{TBool}_F\ f\ (\exists\ e : \mathit{EqP}\ e)$$

A concrete language with boolean and natural types provides a proof algebra of the lemma by composing the proof algebras for the two separate type functors and instantiating the super-functor f to $\mathit{TNat}_F \oplus \mathit{TBool}_F$. By instantiating f to other functor compositions, the proof algebras of the individual features can easily be reused in other languages.

2.4 No Effect Modularity

Unfortunately, effect modularity is not supported in MTC. Consider two features: mutable references Ref_F and errors Err_F. Both of these introduce an effect to any language, the former state and the latter the possibility of raising an error. These effects show up in the type of their evaluation algebras:

$$\begin{aligned}&\mathit{eval}_{\mathit{Ref}} :: \mathit{Algebra}_M\ \mathit{Ref}_F\ (\mathit{Env} \to (\mathit{Value}, \mathit{Env}))\\&\mathit{eval}_{\mathit{Err}} :: \mathit{Algebra}_M\ \mathit{Err}_F\ (\mathit{Maybe}\ \mathit{Value})\end{aligned}$$

MTC supports the composition of two algebras over different functors as long as they have the same carrier. That is not the case here, making the two algebras incompatible. This problem can be solved by anticipating both effects in both algebras:

$$\begin{aligned}&\mathit{eval}_{\mathit{Ref}} :: \mathit{Algebra}_M\ \mathit{Ref}_F\ (\mathit{Env} \to (\mathit{Maybe}\ \mathit{Value}, \mathit{Env}))\\&\mathit{eval}_{\mathit{Err}} :: \mathit{Algebra}_M\ \mathit{Err}_F\ (\mathit{Env} \to (\mathit{Maybe}\ \mathit{Value}, \mathit{Env}))\end{aligned}$$

This anticipation is problematic for modularity: the algebra for references mentions the effect of errors even though it does not involve them, while a language that includes references does not necessarily feature errors. More importantly, the two algebras cannot be composed with a third feature that introduces yet another effect (e.g., local environments) without further foresight. It is impossible to know in advance all the effects that new features may introduce.

3. The 3MT Monad Library

3MT includes a monad library to support effectful semantic functions using *monads* and *monad transformers*, and provides *algebraic laws* for reasoning. Monads provide a uniform representation for encapsulating computational effects such as mutable state, exception handling, and non-determinism. Monad transformers allow monads supporting the desired set of effects to be built. Algebraic laws are the key to *modular* reasoning about monadic definitions.

3MT implements the necessary definitions of *monads* and *monad transformers* as a Coq library inspired by the Haskell *monad transformer library* (MTL) [28]. Our library refines the MTL in two key ways in order to support modular reasoning using algebraic laws. While algebraic laws can only be documented informally in

— Monad class —

class $Functor\ m \Rightarrow Monad\ m$ **where**

$return \quad :: a \to m\ a$

$(\gg\!=) \quad :: m\ a \to (a \to m\ b) \to m\ b$

return_bind $:: return\ x \gg\!= f \equiv f\ x$

bind_return $:: p \gg\!= return \equiv p$

bind_bind $:: (p \gg\!= f) \gg\!= g \equiv p \gg\!= \lambda x \to (f\ x \gg\!= g)$

fmap_bind $:: fmap\ f\ t \equiv t \gg\!= (return \circ f)$

— Failure class —

class $Monad\ m \Rightarrow \mathbb{F}_{\mathrm{M}}\ m$ **where**

$fail \quad :: m\ a$

bind_fail $:: fail \gg\!= f \equiv fail$

— State class —

class $Monad\ m \Rightarrow \mathbb{S}_{\mathrm{M}}\ s\ m$ **where**

$get \quad :: m\ s$

$put \quad :: s \to m\ ()$

get_query $:: get \gg t \equiv t$

put_get $:: put\ s \gg get \equiv put\ s \gg return\ s$

get_put $:: get \gg\!= put \equiv return\ ()$

get_get $:: get \gg\!= \lambda s \gg\!= get \gg\!= f\ s \equiv get \gg\!= \lambda s \to f\ s\ s$

put_put $:: put\ s_1 \gg put\ s_2 \equiv put\ s_2$

— Reader class —

class $Monad\ m \Rightarrow \mathbb{R}_{\mathrm{M}}\ e\ m$ **where**

$ask \quad :: m\ e$

$local \quad :: (e \to e) \to m\ a \to m\ a$

ask_query $:: ask \gg t \equiv t$

local_return $:: local\ f \circ return = return$

ask_ask $:: ask \gg\!= \lambda s \gg\!= ask \gg\!= f\ s \equiv ask \gg\!= \lambda s \to f\ s\ s$

ask_bind $:: t \gg\!= \lambda x \to ask \gg\!= \lambda e \to f\ x\ e \equiv ask \gg\!= \lambda e \to t \gg\!= \lambda x \to f\ x\ e$

local_bind $:: local\ f\ (t \gg\!= g) \equiv local\ f\ t \gg\!= local\ f \circ g$

local_ask $:: local\ f\ ask \equiv ask \gg\!= return \circ f$

local_local $:: local\ f \circ local\ g \equiv local\ (g \circ f)$

— Exception class —

class $Monad\ m \Rightarrow \mathbb{E}_{\mathrm{M}}\ x\ m$ **where**

$throw \quad :: x \to m\ a$

$catch \quad :: m\ a \to (x \to m\ a) \to m\ a$

bind_throw $:: throw\ e \gg\!= f \equiv throw\ e$

catch_throw$_1$ $:: catch\ (throw\ e)\ h \equiv h\ e$

catch_throw$_2$ $:: catch\ t\ throw \equiv t$

catch_return $:: catch\ (return\ x)\ h \equiv return\ x$

fmap_catch $:: fmap\ f\ (catch\ t\ h) \equiv catch\ (fmap\ f\ t)\ (fmap\ f \circ h)$

— Identity monad —

newtype $\mathbb{I}\ a$

$\mathbb{I} \quad :: a \to \mathbb{I}\ a$

$run_{\mathbb{I}} :: \mathbb{I}\ a \to a$

— Failure transformer —

newtype $\mathbb{F}_{\mathrm{T}}\ m\ a$

$\mathbb{F}_{\mathrm{T}} \quad :: m\ (Maybe\ a) \to \mathbb{F}_{\mathrm{T}}\ m\ a$

$run_{\mathbb{F}_{\mathrm{T}}} :: \mathbb{F}_{\mathrm{T}}\ m\ a \to m\ (Maybe\ a)$

— State transformer —

newtype $\mathbb{S}_{\mathrm{T}}\ s\ m\ a$

$\mathbb{S}_{\mathrm{T}} \quad :: (s \to m\ (a, s)) \to \mathbb{S}_{\mathrm{T}}\ s\ m\ a$

$run_{\mathbb{S}_{\mathrm{T}}} :: \mathbb{S}_{\mathrm{T}}\ s\ m\ a \to s \to m\ (a, s)$

— Exception transformer —

newtype $\mathbb{E}_{\mathrm{T}}\ x\ m\ a$

$\mathbb{E}_{\mathrm{T}} \quad :: m\ (Either\ x\ a) \to \mathbb{E}_{\mathrm{T}}\ x\ m\ a$

$run_{\mathbb{E}_{\mathrm{T}}} :: \mathbb{E}_{\mathrm{T}}\ x\ m\ a \to m\ (Either\ x\ a)$

— Reader transformer —

newtype $\mathbb{R}_{\mathrm{T}}\ e\ m\ a$

$\mathbb{R}_{\mathrm{T}} :: (e \to m\ a) \to \mathbb{R}_{\mathrm{T}}\ e\ m\ a$

$run_{\mathbb{R}_{\mathrm{T}}} :: \mathbb{R}_{\mathrm{T}}\ e\ m\ a \to e \to m\ a$

Figure 2. Key classes, definitions and laws from 3MT's monadic library. The names of algebraic laws are in bold.

Haskell, our library fully integrates them into type class definitions using Coq's expressive type system. Additionally, 3MT systematically includes laws for all monad subclasses, several of which have not been covered in the functional programming literature before.

Library overview Figure 2 summarizes the library's key classes, definitions and laws. The type class $Monad$ describes the basic interface of monads. The type $m\ a$ denotes computations of type m which produce values of type a when executed. The function $return$ lifts a value of type a into a (pure) computation that simply produces the value. The *bind* function $\gg\!=$ composes a computation $m\ a$ producing values of type a, with a function that accepts a value of type a and returns a computation of type $m\ b$. The function $\gg$ defines a special case of bind that discards the intermediate value:

$$(\gg) :: Monad\ m \Rightarrow m\ a \to m\ b \to m\ b$$
$$ma \gg mb = ma \gg\!= \backslash\_ \to mb$$

The $\mathbf{do}$ notation is syntactic sugar for the *bind* operator: $\mathbf{do}\ \{x \leftarrow f; g\}$ means $f \gg\!= \lambda x \to g$.

Particular monads can be built from basic monad types such as the identity monad ($\mathbb{I}$) and monad transformers including the failure ($\mathbb{F}_{\mathrm{T}}$), mutable state ($\mathbb{S}_{\mathrm{T}}$), and exception ($\mathbb{E}_{\mathrm{T}}$) transformers. These transformers are combined into different monad stacks with $\mathbb{I}$ at the bottom. Constructor and extractor functions such as $\mathbb{S}_{\mathrm{T}}$ and $run_{\mathbb{S}_{\mathrm{T}}}$ provide the signatures of the functions for building and running particular monads/transformers.

In order to support extensible effects, a feature needs to abstract over the monad implementation used. Any implementation which includes the required operations is valid. These operations are captured in type classes such as $\mathbb{S}_{\mathrm{M}}$ and $\mathbb{E}_{\mathrm{M}}$, also called *monad subclasses*. The type classes (denoted by subscript M) are used to require a monad stack to support a particular effect without assuming a particular stack configuration.[1] Each class offers a set of primitive operations, such as get to access the state for $\mathbb{S}_{\mathrm{M}}$.

Algebraic laws Each monad (sub)class includes a set of algebraic laws that govern its operations. These laws are an integral part of the definition of the monad type classes and constrain the possible implementations to sensible ones. Thus, even without knowing the particular implementation of a type class, we can still modularly reason about its behavior via these laws. This is crucial for supporting modular reasoning [35].

The first three laws for the $Monad$ class are standard, while the last law (***fmap_bind***) relates $fmap$ and $bind$ in the usual way. Each monad subclass also includes its own set of laws. The laws for various subclasses can be found scattered throughout the functional programming literature, such as for failure [13] and state [13, 35]. Yet, as far as we know, 3MT is the first to systematically bring them together. Furthermore, although most laws have been presented in the semantics literature in one form or another, we have not seen some of the laws in the functional programming literature. One such example are the laws for the exception class:

- The ***bind_throw*** law generalizes the ***bind_fail*** law: a sequential computation is aborted by throwing an exception.
- The ***catch_throw***$_1$ law states that the exception handler is invoked when an exception is thrown in a $catch$.
- The ***catch_throw***$_2$ law indicates that an exception handler is redundant if it just re-throws the exception.
- The ***catch_return*** law states that a $catch$ around a pure computation is redundant.
- The ***fmap_catch*** law states that pure functions ($fmap\ \ f$) distribute on the right with $catch$.

Other definitions Our monad library contains a number of other classes, definitions and laws apart from the definitions discussed here. This includes infrastructure for other types of effects (e.g. writer effects), as well as other infrastructure from the MTL. There are roughly 30 algebraic laws in total.

4. Modular Monadic Semantics

Features can utilize the monad library included with 3MT to construct algebras for semantic functions which are compatible with a range of effects. These modular monadic algebras have the form:

$$eval_{Ref} :: \mathbb{S}_{\mathrm{M}}\ Store\ m \Rightarrow Algebra_M\ Ref_F\ (m\ a)$$
$$eval_{Err} :: \mathbb{E}_{\mathrm{M}}\ ()\ m \Rightarrow Algebra_M\ Err_F\ (m\ a)$$

These algebras use monad subclasses such as $\mathbb{S}_{\mathrm{M}}$ and $\mathbb{E}_{\mathrm{M}}$ to *constrain* the monad required by the feature, allowing the monad to

[1] Supporting two instances of the same effect requires extra machinery [41].

```
— Simplified value interface —
type Value

loc   :: Int → Value
stuck :: Value
unit  :: Value
isLoc :: Value → Maybe Int

— Simplified type interface —
type Type

tRef   :: Type → Type
tUnit  :: Type
isTRef :: Type → Maybe Type

—— Expression functor ——
data Ref_F a = Ref a
  | DeRef a
  | Assign a a

type Store = [Value]
```

```
——— Monadic typing algebra ———
typeof_Ref :: 𝔽_M m ⇒
  Algebra_M Ref_F (m Type)
typeof_Ref rec (Ref e) =
  do t ← rec e
     return (tRef t)
typeof_Ref rec (DeRef e) =
  do te ← rec e
     maybe fail return (isTRef te)
typeof_Ref rec (Assign e1 e2) =
  do t1 ← rec e1
     case isTRef t1 of
       Nothing → fail
       Just t → do t2 ← rec e2
                   if (t ≡ t2)
                     then return tUnit
                     else fail
```

```
——————— Monadic evaluation algebra ———————
eval_Ref :: 𝕊_M Store m ⇒ Algebra_M Ref_F (m Value)
eval_Ref rec (Ref e) =
  do v   ← rec e
     env ← get
     put (v : env)
     return (loc (length env))
eval_Ref rec (DeRef e) =
  do v   ← rec e
     env ← get
     case isLoc v of
       Nothing → return stuck
       Just n → return (maybe stuck id (fetch n env))
eval_Ref rec (Assign e1 e2) =
  do v   ← rec e1
     env ← get
     case isLoc v of
       Nothing → return stuck
       Just n   → do v2 ← rec e2
                     put (replace n v2 env)
                     return unit
```

Figure 3. Syntax, type, and semantic function definitions for references.

Arithmetic Expressions	*Monad* m
Boolean Expressions	*Monad* m
Errors	$\mathbb{E}_{\text{M}}$ () m
References	$\mathbb{S}_{\text{M}}$ *Store* m
Lambda	$\mathbb{R}_{\text{M}}$ *Env* m, $\mathbb{F}_{\text{M}}$ m

Figure 4. Effects used by the case study's evaluation algebras.

have more effects than those used in the feature. These two algebras can be combined to create a new evaluation algebra with type:

$$(\mathbb{S}_{\text{M}}\ m\ s, \mathbb{E}_{\text{M}}\ m\ x) \Rightarrow \mathit{Algebra}_M\ (\mathit{Ref}_F \oplus \mathit{Err}_F)\ (m\ a)$$

The combination imposes both type class constraints while the monad type remains extensible with new effects. The complete set of effects used by the evaluation functions for the five language features used in our case study of Section 7 are given in Figure 4.

4.1 Example: References

Figure 3 illustrates this approach with definitions for the functor for expressions and the evaluation and typing algebras for the reference feature. Other features have similar definitions.

For the sake of presentation the definitions are slightly simplified from the actual ones in Coq. For instance, we have omitted issues related to the extensibility of the syntax for values (*Value*) and types (*Type*). We refer the interested reader to MTC [9] and the 3MT Coq code for these details. *Value* and *Type* are treated as abstract datatypes with a number of constructor functions: *loc*, *stuck*, *unit*, *tRef* and *tUnit* denote respectively reference locations, stuck values, unit values, reference types and unit types. There are also matching functions *isLoc* and *isTRef* for checking whether a term is a location value or a reference type, respectively.

The type Ref_F is the functor for references. It has constructors for creating references (*Ref*), dereferencing (*DeRef*) and assigning (*Assign*) references. The evaluation algebra $\mathit{eval}_{\mathit{Ref}}$ uses the state monad for its reference environment, which is captured in the type class constraint $\mathbb{S}_{\text{M}}\ \mathit{Store}\ m$. The typing algebra ($\mathit{typeof}_{\mathit{Ref}}$) is also monadic, using the failure monad to denote ill-typing.

4.2 Effect-Dependent Theorems

Monadic semantic function algebras are compatible with new effects and algebraic laws facilitate writing extensible proofs over these monadic algebras. Effects introduce further challenges to proof reuse, however: each combination of effects induces its own type soundness statement. Consider the theorem for a language with references which features a store σ and a store typing Σ that are related through the store typing judgement $\Sigma \vdash \sigma$:

$$\forall e, t, \Sigma, \sigma. \left\{ \begin{array}{c} \mathit{typeof}\ e \equiv \mathit{return}\ t \\ \Sigma \vdash \sigma \end{array} \right\} \rightarrow$$
$$\exists v, \Sigma', \sigma'. \left\{ \begin{array}{c} \mathit{put}\ \sigma \gg [\![e]\!] \equiv \mathit{put}\ \sigma' \gg \mathit{return}\ v \\ \Sigma' \supseteq \Sigma \\ \Sigma' \vdash v : t \\ \Sigma' \vdash \sigma' \end{array} \right\} \qquad (\text{LSOUND}_S)$$

Contrast this with the theorem for a language with errors, which must account for the computation possibly ending in an exception being thrown:

$$\forall e, t. \mathit{typeof}\ e \equiv \mathit{return}\ t \rightarrow$$
$$(\exists v. [\![e]\!] \equiv \mathit{return}\ v \wedge \vdash v : t) \vee (\exists x. [\![e]\!] \equiv \mathit{throw}\ x) \qquad (\text{LSOUND}_E)$$

Clearly, the available effects are essential for the formulation of the theorem. A larger language which involves both exceptions and state requires yet another theorem where the impact of both effects cross-cut one another[2]:

$$\forall e, t, \Sigma, \sigma. \left\{ \begin{array}{c} \mathit{typeof}\ e \equiv \mathit{return}\ t \\ \Sigma \vdash \sigma \end{array} \right\} \rightarrow$$
$$\exists v, \Sigma', \sigma'. \left\{ \begin{array}{c} \mathit{put}\ \sigma \gg [\![e]\!] \equiv \mathit{put}\ \sigma' \gg \mathit{return}\ v \\ \Sigma' \supseteq \Sigma \\ \Sigma' \vdash v : t \\ \Sigma' \vdash \sigma' \end{array} \right\}$$
$$\vee$$
$$\exists x. \mathit{put}\ \sigma \gg [\![e]\!] \equiv \mathit{throw}\ x \qquad (\text{LSOUND}_{ES})$$

Modular formulations of LSOUND_E and LSOUND_S are useless for proving a modular variant of LSOUND_{ES} because their

[2] A similar proliferation of soundness theorems can be found in TAPL [37].

$$\frac{}{\Sigma \vdash_M v_m \ : \ \mathit{fail}} \quad (\text{WFM-ILLTYPED})$$

$$\frac{\Sigma \vdash v \ : \ t}{\Sigma \vdash_M \mathit{return}\ v \ : \ \mathit{return}\ t} \quad (\text{WFM-RETURN})$$

Figure 5. Typing rules for pure monadic values.

induction hypotheses have the wrong form. The hypothesis for LSOUND$_E$ requires the result to be of the form $\mathit{return}\ v$, disallowing $\mathit{put}\ \sigma' \gg \mathit{return}\ v$ (the form required by LSOUND$_S$). Similarly, the hypothesis for LSOUND$_S$ does not account for exceptions occurring in subterms. In general, without anticipating additional effects, type soundness theorems with fixed sets of effects cannot be reused modularly.

5. Modular Monadic Type Soundness

In order to preserve a measure of modularity, we do not prove type soundness directly for a given feature, but by means of a more generic theorem. The technique of proving a theorem of interest by means of a more general theorem is well-known. For a conventional monolithic language, for instance, type soundness is often established for any well-formed typing context, even though the main interest lies with the more specific initial, empty context. In that setting, the more general theorem produces a weaker induction hypothesis for the theorem's proof.

Our approach to type soundness follows the core idea of this technique and relies on three theorems:

FSOUND: a reusable *feature theorem* that is only aware of the effects that a feature uses

ESOUND: an *effect theorem* for a fixed set of known effects, and

LSOUND: a *language theorem* which combines the two to prove soundness for a specific language.

In order to maximize compatibility, the statement of the reusable feature theorem cannot hardwire the set of effects. This statement must instead rephrase type soundness in a way that can adapt to any superset of a feature's effects. Our solution is to have the feature theorem establish that the monadic evaluation and typing algebras of a feature satisfy an extensible well-formedness relation, defined in terms of effect-specific typing rules. Thus, a feature's proof of FSOUND uses only the typing rules required for the effects specific to that feature. The final language combines the typing rules of all the language's effects into a closed relation.

Figure 6 illustrates how these reusable pieces fit together to build a proof of soundness. Each feature provides a proof algebra for FSOUND which relies on the typing rules (WFM-X) for the effects it uses. Each unique statement of soundness for a combination of effects requires a new proof of ESOUND. The proof of LSOUND for a particular language is synthesized entirely from a single proof of ESOUND and a combination of proof algebras for FSOUND.

Note that there are several dimensions of modularity here. A feature's proof of FSOUND only depends on the typing rules for the effects that feature uses and can thus be used in any language which includes those typing rules. The typing rules themselves can be reused by any number of different features. ESOUND depends solely on a specific combination of effects and can be reused in any language which supports that unique combination, e.g. both LSOUND$_A$ and LSOUND$_{AR}$ use ESOUND$_{ES}$.

5.1 Soundness for a Pure Feature

The reusable feature theorem FSOUND states that $[\![\cdot]\!]$ and *typeof* are related by the extensible typing relation:

$$\forall e, \Sigma. \quad \Sigma \vdash_M [\![e]\!] : \mathit{typeof}\ e \qquad (\text{FSOUND})$$

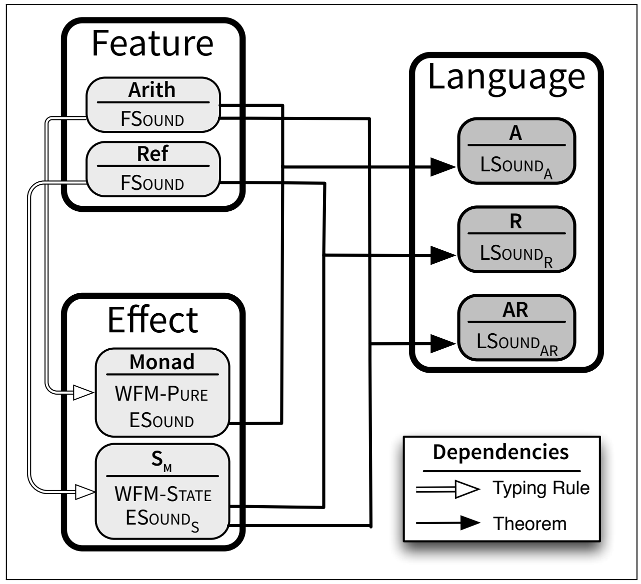

Figure 6. Dependency Graph

Extensible Typing Relation The extensible typing relation has the form:

$$\Sigma \vdash_M v_m : t_m$$

The relation is polymorphic in an environment type *env* and an evaluation monad type m. The parameters Σ, v_m and t_m have types *env*, $m\ \mathit{Value}$ and $\mathit{Maybe}\ \mathit{Type}$ respectively. The modular typing rules for this relation can impose constraints on the environment type *env* and monad type m. A particular language must instantiate *env* and m in a way that satisfies all the constraints imposed by the typing rules used in its features.

Figure 5 lists the two base typing rules of this relation. These do not constrain the evaluation monad and environment types and are the only rules needed for pure features. The (WFM-ILLTYPED) rule denotes that nothing can be said about computations (m_e) which are ill-typed. The (WFM-RETURN) rule ensures that well-typed computations only yield values of the expected type.

To see how the reusable theorem works for a pure feature, consider the proof of soundness for the boolean feature.

Proof Using the above two rules, we can show that FSOUND holds for the boolean feature. The proof has two cases. The boolean literal case is handled by a trivial application of (WFM-RETURN). The second case, for conditionals, is more interesting[3].

$$(\vdash_M [\![e_c]\!] \ : \ \mathit{typeof}\ e_c) \rightarrow$$
$$(\vdash_M [\![e_t]\!] \ : \ \mathit{typeof}\ e_t) \rightarrow$$
$$(\vdash_M [\![e_e]\!] \ : \ \mathit{typeof}\ e_e) \rightarrow$$
$$\vdash_M \begin{pmatrix} \textbf{do}\ v \leftarrow [\![e_c]\!] \\ \quad \textbf{case}\ \mathit{isBool}\ v\ \textbf{of} \\ \quad\quad \mathit{Just}\ b \quad \rightarrow \\ \quad\quad\quad \textbf{if}\ b\ \textbf{then}\ [\![e_t]\!] \\ \quad\quad\quad\quad \textbf{else}\ [\![e_e]\!] \\ \quad\quad \mathit{Nothing} \rightarrow \mathit{stuck} \end{pmatrix} : \begin{pmatrix} \textbf{do}\ t_c \leftarrow \mathit{typeof}\ e_c \\ \quad t_t \leftarrow \mathit{typeof}\ e_t \\ \quad t_e \leftarrow \mathit{typeof}\ e_e \\ \quad \mathit{guard}\ (\mathit{isTBool}\ t_c) \\ \quad \mathit{guard}\ (\mathit{eqT}\ t_t\ t_e) \\ \quad \mathit{return}\ t_t \end{pmatrix}$$

(WFM-IF-VC)

Because $[\![\cdot]\!]$ and *typeof* are polymorphic in the monad, we cannot directly inspect the values they produce. We can, how-

[3] We omit the environment Σ to avoid clutter.

ever, perform case analysis on the derivations of the proofs produced by the induction hypothesis that the subexpressions are well-formed, $\vdash_M [\![e_c]\!] : \mathit{typeof}\ \ e_c$, $\vdash_M [\![e_t]\!] : \mathit{typeof}\ \ e_t$, and $\vdash_M [\![e_e]\!] : \mathit{typeof}\ \ e_e$. The final rule used in each derivation determines the shape of the monadic value produced by $[\![\cdot]\!]$ and typeof. Assuming that only the pure typing rules of Figure 5 are used for the derivations, we can divide the proof into two cases depending on whether e_c, e_t, or e_e was typed with (WFM-ILLTYPED).

- If any of the three derivations uses (WFM-ILLTYPED), the result of *typeof* is *fail*. As *fail* is the zero of the typing monad, (WFM-ILLTYPED) resolves the case.
- Otherwise, each of the subderivations was built with (WFM-RETURN) and the evaluation and typing expressions can be simplified using the ***return_bind*** monad law.

$$\vdash_M \begin{pmatrix} \mathbf{case}\ \mathit{isBool}\ v_c\ \mathbf{of} \\ \quad \mathit{Just}\ b \quad \to \\ \qquad \mathbf{if}\ b\ \mathbf{then}\ \mathit{return}\ v_t \\ \qquad\quad \mathbf{else}\ \mathit{return}\ v_e \\ \quad \mathit{Nothing} \to \mathit{stuck} \end{pmatrix} : \begin{pmatrix} \mathbf{do}\ \mathit{guard}\ (\mathit{isTBool}\ t_c) \\ \quad \mathit{guard}\ (\mathit{eqT}\ t_t\ t_e) \\ \quad \mathit{return}\ t_t \end{pmatrix}$$

After simplification, the typing expression has replaced the bind with explicit values which can be reasoned with. If $\mathit{isTBool}\ \ t_c$ is *false*, then the typing expression reduces to *fail* and well-formedness again follows from the WFM-ILLTYPED rule. Otherwise $t_c \equiv \mathit{TBool}$, and we can apply the inversion lemma

$$\vdash v : \mathit{TBool} \to \exists b. \mathit{isBool}\ v \equiv \mathit{Just}\ b$$

to establish that v_c is of the form $\mathit{Just}\ \ b$, reducing the evaluation to either $\mathit{return}\ \ v_e$ or $\mathit{return}\ \ v_t$. A similar case analysis on $\mathit{eqT}\ \ t_t\ \ t_e$ will either produce *fail* or $\mathit{return}\ \ t_t$. The former is trivially true, and both $\vdash_M\ \ \mathit{return}\ \ v_t\ \ :\ \mathit{return}\ \ t_t$ and $\vdash_M\ \mathit{return}\ \ v_e : \mathit{return}\ \ t_t$ hold in the latter case from the induction hypotheses.

Modular Sublemmas The above proof assumed that only the pure typing rules of Figure 5 were used to type the subexpressions of the **if** expression, which is clearly not the case when the boolean feature is included in an effectful language. Instead, case analyses are performed on the extensible typing relation in order to make the boolean feature theorem compatible with new effects. Case analyses over the extensible $\vdash_M$ relation are accomplished using extensible proof algebras which are folded over the derivations provided by the induction hypothesis, as outlined in Section 2.3.

In order for the boolean feature's proof of FSOUND to be compatible with a new effect, each extensible case analysis requires a proof algebra for the new typing rules the effect introduces to the $\vdash_M$ relation. These proof algebras are examples of *feature interactions* [3] from the setting of 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. Importantly, these proof algebras do not need to be provided up front when developing the boolean algebra, but can instead be modularly resolved by a separate feature for the interaction of booleans and the new effect.

The formulation of the properties proved by extensible case analysis has an impact on modularity. The conditional case of the previous proof can be dispatched by folding a proof algebra for the property WFM-IF-VC over $\vdash_M\ \ [\![v_c]\!]\ \ :\ \ \mathit{typeof}\ \ t_c$. Each new effect induces a new case for this proof algebra, however, resulting in an interaction between booleans and every effect. WFM-IF-VC is specific to the proof of FSOUND in the boolean feature; proofs of FSOUND for other features require different properties and thus different proof algebras. Relying on such specific properties can lead to a proliferation of proof obligations for each new effect.

Alternatively, the boolean feature can use a proof algebra for a stronger property that is also applicable in other proofs, cutting down on the number of feature interactions. One such stronger, more general sublemma relates the monadic bind operation to well-typing:

$$\begin{aligned} &(\Sigma \vdash_M\ \ v_m\ :\ t_m) \to \\ &(\forall v\, T\, \Sigma' \supseteq \Sigma.\ (\Sigma' \vdash\ v\ :\ T) \to\ \Sigma' \vdash_M\ \ k_v\ v\ :\ k_t\ T) \to \\ &\qquad\qquad \Sigma \vdash_M\ \ v_m \gg\!= k_v\ :\ t_m \gg\!= k_t \qquad \text{(WFM-BIND)} \end{aligned}$$

A proof of WFM-IF-VC follows from two applications of this stronger property. The advantage of WFM-BIND is clear: it can be reused to deal with case analyses in other proofs of FSOUND, while a proof of WFM-IF-VC has only a single use. The disadvantage is that WFM-BIND may not hold for some new effect, while the weaker WFM-IF-VC does, possibly excluding some feature combinations. As WFM-BIND is a desirable property for typing rules, the case study focuses on that approach.

5.2 Type Soundness for a Pure Language

The second phase of showing type soundness is to prove a statement of soundness for a fixed set of effects. For pure effects, the soundness statement is straightforward:

$$\forall v_m\ t. \vdash_M v_m : \mathit{return}\ t \Rightarrow \exists v. v_m \equiv \mathit{return}\ v \wedge \vdash v : t \qquad (\text{ESOUND}_P)$$

Each effect theorem is proved by induction over the derivation of $\vdash_M\ v_m : \mathit{return}\ \ t$. ESOUND_P fixes the irrelevant environment type to the type $()$ and the evaluation monad to the pure monad $\mathbb{I}$. Since the evaluation monad is fixed, the proof of ESOUND_P only needs to consider the pure typing rules of Figure 5. The proof of the effect theorem is straightforward: WFM-ILLTYPED could not have been used to derive $\vdash_M\ v_m : \mathit{return}\ \ t$, and WFM-RETURN provides both a witness for v and a proof that it is of type t.

The statement of soundness for a pure language built from a particular set of features is similar to ESOUND_P:

$$\forall e, t. \mathit{typeof}\ \ e \equiv \mathit{return}\ t \Rightarrow \exists v. [\![e]\!] \equiv \mathit{return}\ v \wedge \vdash v : t \qquad (\text{LSOUND})$$

The proof of LSOUND is an immediate consequence of the reusable proofs of FSOUND and ESOUND_P. Folding a proof algebra for FSOUND over e provides a proof of $\vdash_M [\![e]\!] : \mathit{return}\ \ t$, satisfying the first assumption of ESOUND_P. LSOUND follows immediately.

5.3 Errors

The evaluation algebra of the error language feature uses the side effects of the exception monad, requiring new typing rules.

$$\frac{}{\Sigma \vdash_M \mathit{throw}\ x\ :\ t_m} \qquad \text{(WFM-THROW)}$$

$$\frac{\Sigma \vdash_M m \gg\!= k\ :\ t_m \qquad \forall\, \Sigma' \supseteq \Sigma\ x\ .\ \Sigma' \vdash_M h\ x \gg\!= k\ :\ t_m}{\Sigma \vdash_M \mathit{catch}\ m\ h \gg\!= k\ :\ t_m} \qquad \text{(WFM-CATCH)}$$

Figure 7. Typing rules for exceptional monadic values.

Typing Rules Figure 7 lists the typing rules for monadic computations involving exceptions. WFM-THROW states that $\mathit{throw}\ x$ is typeable with any type. WFM-CATCH states that binding the results of both branches of a *catch* statement will produce a monad with the same type. While it may seem odd that this rule is formulated in terms of a continuation $\gg\!= k$, it is essential for compatibility with the proofs algebras required by other features. As described

$$\frac{\forall \sigma, \Sigma \vdash \sigma \;\rightarrow\; \Sigma \vdash_M k\,\sigma : t_m}{\Sigma \vdash_M \mathit{get} \ggg k \;:\; t_m} \quad (\text{WFM-GET})$$

$$\frac{\Sigma' \vdash \sigma \qquad \Sigma' \supseteq \Sigma \qquad \Sigma' \vdash_M k : t_m}{\Sigma \vdash_M \mathit{put}\ \sigma \gg k \;: t_m} \quad (\text{WFM-PUT})$$

Figure 8. Typing rules for stateful monadic values.

in Section 5.1, extensible proof algebras over the typing derivation will now need cases for the two new rules. To illustrate this, consider the proof algebra for the general purpose WFM-BIND property. This algebra requires a proof of:

$$\begin{aligned}
&(\Sigma \vdash_M \mathit{catch}\ e\ h \ggg k \;:\; t_m) \rightarrow \\
&(\forall v\, T\, \Sigma' \supseteq \Sigma.\, (\Sigma' \vdash\ v \;:\; T) \rightarrow\ \Sigma' \vdash_M\ \ k_v\ v \;:\; k_t\ T) \rightarrow \\
&\qquad\qquad \Sigma \vdash_M (\mathit{catch}\ e\ h \ggg k) \ggg k_v : t_m \ggg k_t
\end{aligned}$$

With the continuation, we can first apply the associativity law to reorder the binds so that WFM-CATCH can be applied: $(\mathit{catch}\ e\ h \ggg k) \ggg k_v = \mathit{catch}\ e\ h \ggg (k \ggg k_v)$. The two premises of the rule follow immediately from the inductive hypothesis of the lemma, finishing the proof. Without the continuation, the proof statement only binds $\mathit{catch}\ e\ h$ to v_m, leaving no applicable typing rules.

Effect Theorem The effect theorem, ESOUND_E, for a language whose only effect is exceptions reflects that the evaluation function is either a well-typed value or an exception.

$$\begin{aligned}
&\forall v_m\ t. \vdash_M\ v_m : \mathit{return}\ t \Rightarrow \\
&\exists x.v_m \equiv \mathit{throw}\ x \vee \exists v.v_m \equiv \mathit{return}\ v \wedge \vdash\ v : t \quad (\text{ESOUND}_E)
\end{aligned}$$

The proof of ESOUND_E is again by induction on the derivation of $\vdash_M v_m : \mathit{return}\ t$. The irrelevant environment can be fixed to (), while the evaluation monad is the exception monad $\mathbb{E}_{\mathrm{T}}\ x\ \mathbb{I}$.

The typing derivation is built from four rules: the two pure rules from Figure 5 and the two exception rules from Figure 7. The case for the two pure rules is effectively the same as before, and WFM-THROW is straightforward. In the remaining case, $v_m \equiv \mathit{catch}\ e'\ h$, and we can leverage the fact that the evaluation monad is fixed to conclude that either $\exists v.e' \equiv \mathit{return}\ v$ or $\exists x.e' \equiv \mathit{throw}\ x$. In the former case, $\mathit{catch}\ e'\ h$ can be reduced using ***catch_return***, and the latter case is simplified using ***catch_throw***$_1$. In both cases, the conclusion then follows immediately from the assumptions of WFM-CATCH. The proof of the language theorem LSOUND_E is similar to LSOUND and is easily built from ESOUND_E and FSOUND.

5.4 References

Typing Rules Figure 8 lists the two typing rules for stateful computations. To understand the formulation of these rules, consider LSOUND_S, the statement of soundness for a language with a stateful evaluation function. The statement accounts for both the typing environment Σ and evaluation environment σ by imposing the invariant that σ is well-formed with respect to Σ. FSOUND however, has no such conditions (which would be anti-modular in any case). We avoid this problem by accounting for the invariant in the typing rules themselves:

- WFM-GET requires that the continuation k of a get is well-typed under the invariant.
- WFM-PUT requires that any newly installed environment maintains this invariant.

The intuition behind these premises is that effect theorems will maintain these invariants in order to apply the rules.

$$\frac{\forall \gamma.\, \Gamma \vdash \gamma \;\rightarrow\; \Gamma \vdash_M k\,\gamma : t_m}{\Gamma \vdash_M \mathit{ask} \ggg k \;:\; t_m} \quad (\text{WFM-ASK})$$

$$\frac{\begin{array}{c}\forall\, \gamma.\, \Gamma \vdash \gamma \rightarrow \Gamma' \vdash f\ \gamma \qquad \Gamma' \vdash_M m \;:\; \mathit{return}\ \ t'_m \\ \forall v.\ \vdash v \;:\; t'_m \rightarrow \Gamma \vdash_M (k\ v) : t_m\end{array}}{\Gamma \vdash_M\ \ \mathit{local}\ f\ m \ggg k\!:\ t_m} \quad (\text{WFM-LOCAL})$$

$$\frac{}{\Gamma \vdash_M \bot \;:\; t_m} \quad (\text{WFM-BOT})$$

Figure 9. Typing rules for environment and failure monads.

Effect Theorem The effect theorem for mutable state proceeds again by induction over the typing derivation. The evaluation monad is fixed to $\mathbb{S}_{\mathrm{T}}\ \mathit{Sigma}\ \mathbb{I}$ and the environment type is fixed to $[\,\mathit{Type}]$ with the obvious definitions for $\supseteq$.

- The proof case for the two pure rules is again straightforward.
- For WFM-GET we have that $\mathit{put}\ \sigma \gg [\![e]\!] \equiv \mathit{put}\ \sigma \gg \mathit{get} \ggg k$. After reducing this to $k\ \sigma$ with the ***put_get*** law, the result follows immediately from the rule's assumptions.
- Similarly, for WFM-PUT we have that $\mathit{put}\ \sigma \gg [\![e]\!] \equiv \mathit{put}\ \sigma \gg \mathit{put}\ \sigma' \gg k$. After reducing this to $\mathit{put}\ \sigma' \gg k$ with the ***put_put*** law, the result again follows immediately from the rule's assumptions.

5.5 Lambda

The case study represents the binders of the lambda feature using PHOAS [7] to avoid many of the boilerplate definitions and proofs about term well-formedness found in first-order representations.

The Environment Effect Unlike in MTC, 3MT neatly hides the variable environment of the evaluation function with a reader monad $\mathbb{R}_{\mathrm{M}}$. This new effect introduces the two new typing rules listed in Figure 9. Unsurprisingly, these typing rule are similar to those of Figure 8. The rule for *ask* is essentially the same as WFM-GET. The typing rule for *local* differs slightly from WFM-PUT. Its first premise ensures that whenever f is applied to an environment that is well-formed in the original typing environment Γ, the resulting environment is well-formed in some new environment Γ'. The second premise ensures the body of *local* is well-formed in this environment according to some type T, and the final premise ensures that k is well-formed when applied to any value of type T. The intuition behind binding the *local* expression in some k is the same as with put.

The Non-Termination Effect The lambda feature also introduces the possibility of non-termination to the evaluation function, which is disallowed by Coq. MTC solves this problem by combining *mixin algebras* with a bounded fixpoint function. This function applies an algebra a bounded number of times, returning a $\bot$ value when the bound is exceeded. Because MTC represented $\bot$ as a value, all evaluation algebras needed to account for it explicitly. In the monadic setting, 3MT elegantly represents $\bot$ with the *fail* primitive of the failure monad. This allows terminating features to be completely oblivious to whether a bounded or standard fold is used for the evaluation function, resulting in a much cleaner semantics. WFM-BOT allows $\bot$ to have any type.

6. Effect Compositions

As we have seen, laws are essential for proofs of FSOUND. The proofs so far have involved only one effect and the laws regulate the behavior of that effect's primitive operations.

Languages often involve more than one effect, however. Hence, the proofs of effect theorems must reason about the interaction between multiple effects. There is a trade-off between fully instantiating the monad for the language as we have done previously, and continuing to reason about a constrained polymorphic monad. The former is easy for reasoning, while the latter allows the same language proof to be instantiated with different implementations of the monad. In the latter case, additional *effect interaction* laws are required.

6.1 Languages with State and Exceptions

Consider the effect theorem which fixes the evaluation monad to support exceptions and state. The statement of the theorem mentions both kinds of effects by requiring the evaluation function to be run with a well-formed state σ and by concluding that well-typed expressions either throw an exception or return a value. The WFM-CATCH case this theorem has the following goal:

$$\begin{array}{c} (\Sigma \vdash \sigma \ :\ \Sigma) \\ \rightarrow \\ \exists\, \Sigma', \sigma', v. \left\{ \begin{array}{c} put\ \sigma \gg catch\ e\ h \ggg k \equiv put\ \sigma' \gg return\ v \\ \Sigma' \vdash v : t \end{array} \right\} \\ \vee \\ \exists\, \Sigma', \sigma', x. \left\{ \begin{array}{c} put\ \sigma \gg catch\ e\ h \ggg k \equiv put\ \sigma' \gg throw\ x \\ \Sigma' \vdash \sigma' \ :\ \Sigma' \end{array} \right\} \end{array}$$

In order to apply the induction hypothesis to e and h, we need to precede them by a $put\ \sigma$. Hence, $put\ \sigma$ must be pushed under the $catch$ statement through the use of a law governing the behavior of put and $catch$. There are different choices for this law, depending on the monad that implements both $\mathbb{S}_\text{M}$ and $\mathbb{E}_\text{M}$. We consider two reasonable choices, based on the monad transformer compositions $\mathbb{E}_\text{T}\ x\ (\mathbb{S}_\text{T}\ s\ \mathbb{I})$ and $\mathbb{S}_\text{T}\ s\ (\mathbb{E}_\text{T}\ x\ \mathbb{I})$:

- Either $catch$ passes the current state into the handler:
  $put\ \sigma \gg catch\ e\ h \equiv catch\ (put\ \sigma \gg e)\ h$
- Or $catch$ runs the handler with the initial state:
  $put\ \sigma \gg catch\ e\ h \equiv catch\ (put\ \sigma \gg e)\ (put\ \sigma \gg h)$

The WFM-CATCH case is provable under either choice. As the LSOUND$_{ES}$ proof is written as an extensible theorem, the two cases are written as two separate proof algebras, each with a different assumption about the behavior of the interaction. Since the cases for the other rules are impervious to the choice, they can be reused with either proof of WFM-CATCH.

6.2 Full Combination of Effects

A language with references, errors and lambda abstractions features four effects: state, exceptions, an environment and failure. The language theorem for such a language relies on the effect theorem ESOUND$_{ESRF}$ given in Figure 10. The proof of ESOUND$_{ESRF}$ is similar to the previous effect theorem proofs, and makes use of the full set of interaction laws given in Figure 11. Perhaps the most interesting observation here is that because the environment monad only makes local changes, we can avoid having to choose between laws regarding how it interacts with exceptions. Also note that since we are representing nontermination using a failure monad $\mathbb{F}_\text{M}\ m$, the ***catch_fail*** law conforms to our desired semantics.

7. Case Study

As a demonstration of the 3MT framework, we have built a set of five reusable language features and combined them to build a family of languages which includes a mini-ML [8] variant with references and errors. The study includes pure boolean and arithmetic features as well as effectful features for references, errors and lambda abstractions.

$$\forall \Sigma, \Gamma, \delta, \gamma, \sigma, e_E, e_T. \left\{ \begin{array}{c} \gamma, \delta \vdash\ e_E \equiv e_T \\ \Sigma \vdash \sigma \ :\ \Sigma \\ \Sigma \vdash \gamma \ :\ \Gamma \\ typeof\ \ e_T \equiv return\ t \end{array} \right\} \rightarrow$$

$$\exists\, \Sigma', \sigma', v. \left\{ \begin{array}{c} local\ (\lambda\_.\gamma)\ (put\ \sigma \gg [\![e]\!]_E) \\ \equiv local\ (\lambda\_.\gamma)\ (put\ \sigma' \gg return\ v) \\ \Sigma' \vdash v : t \end{array} \right\}$$

$$\vee$$

$$\exists\, \Sigma', \sigma', v. \left\{ \begin{array}{c} local\ (\lambda\_.\gamma)\ (put\ \sigma \gg [\![e]\!]_E) \\ \equiv local\ (\lambda\_.\gamma(put\ \sigma' \gg \bot) \\ \Sigma' \vdash \sigma' \ :\ \Sigma' \\ \Sigma' \supseteq \Sigma \end{array} \right\}$$

$$\vee$$

$$\exists\, \Sigma', \sigma', v. \left\{ \begin{array}{c} local\ (\lambda\_.\gamma)\ (put\ \sigma \gg [\![e]\!]_E) \\ \equiv local\ (\lambda\_.\Gamma(put\ \sigma' \gg throw\ t) \\ \Sigma' \vdash \sigma' \ :\ \Sigma' \\ \Sigma' \supseteq \Sigma \end{array} \right\}$$

(ESOUND$_{ESRF}$)

Figure 10. Effect theorem statement for languages with errors, state, an environment and failure.

The study builds twenty eight different combinations of the features which are all possible combinations with at least one feature providing values.[4] Figure 13 presents the syntax of the expressions, values, and types provided; each line is annotated with the feature that provides that set of definitions.

Four kinds of feature interactions appear in the case study.

- The PHOAS representation of binders requires an auxiliary equivalence relation, the details of which are covered in the MTC paper [9]. The soundness proofs of language theorems

[4] Also available at `http://www.cs.utexas.edu/~bendy/3MT`

Exceptional Environment

class $(\mathbb{E}_\text{M}\ x\ m, \mathbb{R}_\text{M}\ m) \Rightarrow \mathbb{ER}_\text{M}\ x\ g\ m$ **where**
local_throw $:: local\ f\ (throw\ e) \equiv throw\ e$
local_catch $:: local\ f\ (catch\ e\ h) \equiv$
$catch\ (local\ f\ e)\ (\lambda x.local\ f\ (h\ x))$

Exceptional Failure

class $(\mathbb{E}_\text{M}\ x\ m, \mathbb{F}_\text{M}\ m) \Rightarrow \mathbb{FS}_\text{M}\ x\ m$ **where**
catch_fail $:: catch\ fail\ h \equiv fail$
fail_neq_throw $:: fail \not\equiv throw\ x$

Exceptional State Failure

class $(\mathbb{E}_\text{M}\ x\ m, \mathbb{S}_\text{M}\ s\ m, \mathbb{F}_\text{M}\ m) \Rightarrow \mathbb{EFS}_\text{M}\ x\ m$ **where**
put_fail_throw $:: put\ \sigma \gg fail \not\equiv put\ \sigma' \gg throw\ x$

Exceptional State

class $(\mathbb{E}_\text{M}\ x\ m, \mathbb{F}_\text{M}\ m) \Rightarrow MonadErrorState\ x\ m$ **where**
put_ret_throw $:: put\ \sigma \gg return\ a \not\equiv put\ \sigma' \gg throw\ x$
put_throw $:: \forall A\ B.put\ \sigma \gg throw\_A\ x \equiv put\ \sigma' \gg throw\_A\ x \rightarrow$
$put\ \sigma \gg throw\_B\ x \equiv put\ \sigma' \gg throw\_B\ x$

Alternate Exceptional State laws

class $(\mathbb{E}_\text{M}\ x\ m, \mathbb{F}_\text{M}\ m) \Rightarrow \mathbb{ES}_{\text{M}_1}\ x\ m$ **where**
put_catch$_1$ $:: put\ \sigma \gg catch\ e\ h \equiv catch\ (put\ \sigma \gg e)\ h$

Or

class $(\mathbb{E}_\text{M}\ x\ m, \mathbb{F}_\text{M}\ m) \Rightarrow \mathbb{ES}_{\text{M}_2}\ x\ m$ **where**
put_catch$_2$ $:: put\ env \gg catch\ e\ h \equiv$
$catch\ (put\ \sigma \gg e)\ (\lambda x \rightarrow put\ \sigma \gg h\ x)$

Figure 11. Interaction laws

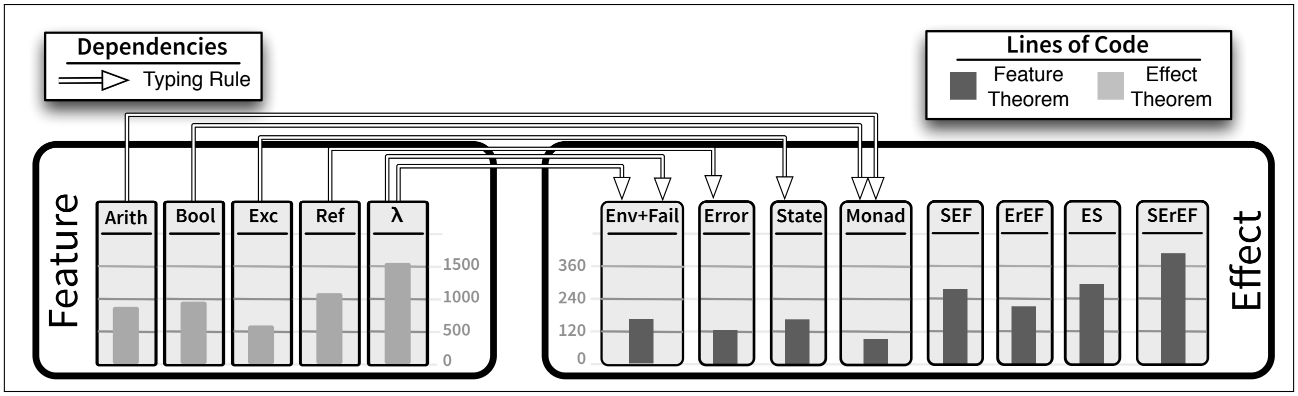

Figure 12. Dependency and size information for the features and effects used in the case study.

$\mathtt{e} ::= \mathbb{N} \mid \mathtt{e} + \mathtt{e}$	*Arith*
$\mid \mathbb{B} \mid$ **if** $\mathtt{e}$ **then** $\mathtt{e}$ **else** $\mathtt{e}$	*Bool*
$\mid$ **lam** $\mathtt{x} : \mathtt{T.e} \mid \mathtt{e}\ \mathtt{e} \mid \mathtt{x}$	*Lambda*
$\mid$ **ref** $\mathtt{e} \mid \mathtt{!e} \mid \mathtt{e{:=}e}$	*References*
$\mid$ **try** $\mathtt{e}$ **with** $\mathtt{e} \mid$ **error**	*Errors*

$\mathtt{V} ::= \mathbb{N}$	*Arith*
$\mid \mathbb{B}$	*Bool*
$\mid$ **clos** $\mathtt{e}\ \overline{\mathtt{V}}$	*Lambda*
$\mid$ **loc** $\mathbb{N}$	*References*

$\mathtt{T} ::=$ **Nat**	*Arith*
$\mid$ **Bool**	*Bool*
$\mid \mathtt{T} \rightarrow \mathtt{T}$	*Lambda*
$\mid$ **Ref** $\mathtt{T}$	*References*

Figure 13. mini-ML expressions, values, and types

for languages which include binders proceed by induction over this equivalence relation instead of expressions. The reusable feature theorems of other features need to be lifted to this equivalence relation.

- The effect theorems that feature an environment typing Σ, such as those for state or environment, need a weakening sublemma which states that each well-formed value under Σ is also well-formed under a conservative extension:

$$\Sigma \vdash v \ : \ t \to \Sigma' \supseteq \Sigma \to \Sigma' \vdash v \ : \ t$$

- Inversion lemmas for the well-formed value relation as in the proof of FSOUND for the boolean feature in Section 5.1 are proven by induction over the relation.

The proofs of the first and second kind of feature interactions are straightforward; the inversion lemmas of the third kind can be dispatched by tactics hooked into the type class inference algorithm.

The framework itself consists of about 4,400 LoC of which about 2,000 LoC comprise the implementation of the monad transformers and their algebraic laws. The size in LoC of the implementation of semantic evaluation and typing functions and the reusable feature theorem for each language feature is given in the left box in Figure 12. The right box lists the sizes of the effect theorems. Each language needs on average 110 LoC to assemble its semantic functions and soundness proofs from those of its features and the effect theorem for its set of effects.

8. Related Work

While previous work has explored the basic techniques of modularizing dynamic semantics of languages with effects, our work is the first to show how to also do modular proofs. Adding the ability to do modular proofs required the development of novel techniques for reasoning about modular components with effects.

8.1 Functional Models for Modular Side Effects

Monads and Monad Transformers Since Moggi [31] first proposed monads to model side-effects, and Wadler [50] popularized them in the context of Haskell, various researchers (e.g., [21, 45]) have sought to modularize monads. Monad transformers emerged [6, 28] from this process, and in later years various alternative implementation designs facilitating monad (transformer) implementations, have been developed, including Filinksi's layered monads [10] and Jaskelioff's Monatron [19].

Monads and Subtyping Filinski's MultiMonadic MetaLanguage (M^3L) [11, 12] embraces the monadic approach, but uses subtyping (or subeffecting) to combine the effects of different components. The subtyping relation is fixed at the program or language level, which does not provide the adaptability we achieve with constrained polymorphism.

Algebraic Effects and Effect Handlers In the semantics community the algebraic theory of computational effects [39] has been an active area of research. Many of the laws about effects, which we have not seen before in the context of functional programming, can be found throughout the semantics literature. Our first four laws for exceptions, for example, have been presented by Levy [26].

A more recent model of side effects are effect handlers. They were introduced by Plotkin and Pretnar [38] as a generalization from exception handlers to handlers for a range of computational effects, such as I/O, state, and nondeterminism. Bauer and Pretnar [4] built the language *Eff* around effect handlers and show how to implement a wide range of effects in it. Kammar et al. [22] showed that effect handlers can be implemented in terms of delimited continuations or free monads.

The major advantage of effect handlers over monads is that they are more easily composed, as any composition of effect operations and corresponding handlers is valid. In contrast, not every composition of monads is a monad. In the future, we plan on investigating the use of effect handlers instead of monad transformers, which could potentially reduce the amount of work involved on proofs about interactions of effects.

Other Effect Models Other useful models have been proposed, such as *applicative functors* [30] and *arrows* [17], each with their own axioms and modularity properties.

8.2 Modular Effectful Semantics

There are several works on how to modularize semantics with effects, although none of these works considers reasoning.

Mosses [33] modularizes structural operational semantics by means of a label transition system where extensible labels capture effects like state and abrupt termination. Swierstra [46] presents modular syntax with functor coproducts and modular semantics with algebra compositions. To support effects, he uses modular syntax to define a free monad. The effectful semantics for this free monad is not given in a modular manner, however. Jaskelioff et al. [20] present a modular approach for operational semantics on top of Swierstra's modular syntax, although they do not cover conventional semantics with side-effects. Both Schrijvers and Oliveira [42] and Bahr and Hvitved [2] have shown how to define modular semantics with monads for effects; this is essentially the approach followed in this paper for modular semantics.

8.3 Effects and Reasoning

Non-Modular Monadic Reasoning Although monads are a purely functional way to encapsulate computational-effects, programs using monads are challenging to reason about. The main issue is that monads provide an abstraction over purely functional models of effects, allowing functional programmers to write programs in terms of abstract operations like $\gg=$, $return$, or get and put. One way to reason about monadic programs is to remove the abstraction provided by such operations [18]. However, this approach is fundamentally non-modular.

Modular Monadic Reasoning Several more modular approaches to modular monadic reasoning have been pursued in the past.

One approach to modular monadic reasoning is to exploit *parametricity* [40, 49]. Voigtländer [48] has shown how to derive parametricity theorems for type constructor classes such as $Monad$. Unfortunately, the reasoning power of parametricity is limited, and parametricity is not supported by proof assistants like Coq.

A second technique uses *algebraic laws*. Liang and Hudak [27] present one of the earliest examples of using algebraic laws for reasoning. They use algebraic laws for reader monads to prove correctness properties about a modular compiler. In contrast to our work, their compiler correctness proofs are pen-and-paper and thus more informal than our proofs. Since they are not restricted by a termination checker or the use of positive types only, they exploit features like general recursion in their definitions. Oliveira et al. [35] have also used algebraic laws for the state monad, in combination with parametricity, for modular proofs of non-interference of aspect-oriented advice. Hinze and Gibbons discuss several other algebraic laws for various types of monads [13]. However, as far as we know, we are the first to provide an extensive mechanized library for monads and algebraic laws in Coq.

8.4 Mechanization of Monad Transformers

Huffmann [16] illustrates an approach for mechanizing type constructor classes in Isabelle/HOL with monad transformers. He considers transformer variants of the resumption, error and writer monads, but features only the generic functor, monad and transformer laws. The work tackles many issues that are not relevant for our Coq setting, such as lack of parametric polymorphism and explicit modeling of laziness.

9. Conclusion

In previous work [9] we have shown that it is possible to modularize meta-theory along two dimensions: 1) language constructs and 2) operations and proofs. A significant limitation of that work is that it only considered pure languages.

This work lifts that limitation and shows how to develop modular meta-theory for languages with effects. Our solution uses monads and corresponding algebraic laws for reasoning about different types of effects. The key challenge that we have solved is how to formulate and prove a general type-soundness theorem in a modular way that enables the reuse of feature proofs across multiple languages with different sets of effects. This turned out to be non-trivial because existing formulations of type-soundness are very sensitive to the particular effects used by the language.

As a secondary contribution, our work shows that algebraic laws about effects scale up to realistic verification tasks such as meta-theoretic proofs. As far as we know, it is their largest application to date. In this setting, the proof assistant Coq has been invaluable. While the typically smaller examples in the functional programming community can easily be dealt with by pen-and-paper proofs, that approach would not have been manageable for the large family of type-soundness proofs for mini-ML variants, as keeping track of large goals and hypotheses by hand would be too painful and error-prone.

Acknowledgements We would like to thank the anonymous reviewers for their many comments and suggestions. This work was supported by the National Science Foundation under Grant CCF 0724979.

References

[1] Brian E. Aydemir et al. Mechanized metatheory for the masses: The poplmark challenge. In *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

[2] Patrick Bahr and Tom Hvitved. Compositional data types. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, WGP '11, pages 83–94. ACM, 2011.

[3] Don Batory, Jongwook Kim, and Peter Höfner. Feature interactions, products, and composition. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11. ACM, 2011.

[4] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *CoRR*, abs/1203.1539, 2012.

[5] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed -programs on term algebras. *Theoretical Computer Science*, 39(0):135 – 154, 1985.

[6] Pietro Cenciarelli and Eugenio Moggi. A syntactic approach to modularity in denotational semantics. In *In Proceedings of the Conference on Category Theory and Computer Science*, CCTCS '93, 1993.

[7] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 143–156. ACM, 2008.

[8] Dominique Clément, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. A simple applicative language: mini-ml. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 13–27. ACM, 1986.

[9] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 207–218. ACM, 2013.

[10] Andrzej Filinski. Representing layered monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 175–188. ACM, 1999.

[11] Andrzej Filinski. On the relations between monadic semantics. *Theor. Comput. Sci.*, 375(1-3):41–75, 2007.

[12] Andrzej Filinski. Monads in action. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 483–494. ACM, 2010.

[13] Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 2–14. ACM, 2011.

[14] Joseph A. Goguen, James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24(1), 1977.

[15] Georges Gonthier. Engineering mathematics: the odd order theorem proof. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 1–2. ACM, 2013.

[16] Brian Huffman. Formal verification of monad transformers. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 15–16. ACM, 2012.

[17] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[18] Graham Hutton and Diana Fulger. Reasoning about effects: Seeing the wood through the trees. In *Proceedings of the Ninth Symposium on Trends in Functional Programming*, 2008.

[19] Mauro Jaskelioff. Monatron: An extensible monad transformer library. In *Implementation and Application of Functional Languages*, volume 5836 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2011.

[20] Mauro Jaskelioff, Neil Ghani, and Graham Hutton. Modularity and implementation of mathematical operational semantics. *Electron. Notes Theor. Comput. Sci.*, 229(5):75–95, 2011.

[21] Mark P. Jones and Luc Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, 1993.

[22] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *The 1st ACM SIGPLAN Workshop on Higher-Order Programming with Effects*, HOPE '12, 2012.

[23] Klein et al. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[24] Peter Lee. *Realistic Compiler Generation*. MIT Press, Cambridge, MA, 1989.

[25] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[26] Paul Blain Levy. Monads and adjunctions for global exceptions. *Electron. Notes Theor. Comput. Sci.*, 158:261–287, 2006.

[27] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *Proceedings of the 6th European Symposium on Programming Languages and Systems*, ESOP '96, pages 219–234. Springer-Verlag, 1996.

[28] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 333–343. ACM, 1995.

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

[30] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[31] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science, June 1989.

[32] Peter D. Mosses. A basic abstract semantic algebra. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 87–107. Springer, 1984.

[33] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 6061(0):195 – 228, 2004.

[34] Bruno C. d. S. Oliveira. Modular visitor components. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, ECOOP 2009, pages 269–293. Springer-Verlag, 2009.

[35] Bruno C. d. S. Oliveira, Tom Schrijvers, and William R. Cook. EffectiveAdvice: disciplined advice with explicit effects. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 109–120. ACM, 2010.

[36] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer-Verlag, 1990.

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

[38] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009.

[39] Gordon D. Plotkin and John Power. Notions of computation determine monads. In *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, FoSSaCS '02, pages 342–356. Springer-Verlag, 2002.

[40] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[41] Tom Schrijvers and Bruno C. d. S. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 32–44. ACM, 2011.

[42] Tom Schrijvers and Bruno C. d. S. Oliveira. The monad zipper. Report CW 595, Dept. of Computer Science, K.U.Leuven, 2010.

[43] Zhong Shao. Certified software. *Commun. ACM*, 53(12):56–66, 2010.

[44] Antonis Stampoulis and Zhong Shao. Veriml: typed computation of logical terms inside a language with effects. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 333–344. ACM, 2010.

[45] Guy L. Steele, Jr. Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 472–492. ACM, 1994.

[46] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.

[47] Tarmo Uustalu and Varmo Vene. Coding recursion à la Mendler. In *Proceedings 2nd Workshop on Generic Programming, WGP '00*, pages 69–85, 2000.

[48] Janis Voigtländer. Free theorems involving type constructor classes: functional pearl. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 173–184. ACM, 2009.

[49] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 347–359. ACM, 1989.

[50] Philip Wadler. Monads for functional programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*, August 1992.

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

[52] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76. ACM, 1989.

[53] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38 – 94, 1994.