The Design of a Practical Proof Checker for a Lazy Functional Language

Adam Procter[1]*, William L. Harrison[1], and Aaron Stump[2]

[1] Dept. of Computer Science, University of Missouri, Columbia, Missouri, USA
[2] Dept. of Computer Science, University of Iowa, Iowa City, Iowa, USA

Abstract. Pure, lazy functional languages like Haskell provide a sound basis for formal reasoning about programs in an equational style. In practice, however, equational reasoning is underutilized. We suggest that part of the reason for this is the lack of accessible tools for developing machine-checked equational reasoning proofs. This paper outlines the design of MProver, a system which fills just that niche. MProver features first-class support for reasoning about potentially undefined computations (particularly important in a lazy setting), and an extended notion of Haskell-like type classes, enabling a highly modular style of program verification that closely follows familiar functional programming idioms.

1 Introduction

The grand promise of pure functional languages is a mathematically rigorous style of programming—a style in which the meaning of a program is defined precisely and compositionally, and program properties may be reasoned about statically according to intuitive yet precise laws. The use of a lazy or non-strict semantics, as exemplified by Haskell, enables a wide array of proof techniques based on the simple unifying principle of *equational reasoning*: if it can be shown that subterm t in a program always evaluates to the same thing as t', we may subsitute t with t' without fear of changing the program's meaning in subtle ways.

The strong, static type system of Haskell is a highly successful example of "lightweight formal methods", capable of detecting and preventing many kinds of programming errors. However, it does not have the power to express, let alone enforce, many useful properties that can be proved via external equational reasoning. Yet there is a gap when it comes to tools: while the Haskell type checker automatically decides whether a program is well typed, few tools in widespread use support automatic checking of equational reasoning proofs. We believe that this lack is a serious obstacle to broader adoption of equational reasoning, and that developing such a tool would make equational reasoning accessible to a wider audience.

This paper describes the design of a new system called *MProver* for proving equational properties of programs in a pure, lazy, functional language. Our main

* Supported by the US Department of Education under grant number P200A100053.

motivation in developing MProver is to support machine-checked equational reasoning proofs about programs with monadic effects (hence the *M* in *MProver*). The system is, however, useful for all kinds of functional programming idioms; it is not limited, nor even specifically tailored, to monadic programs.

In comparing MProver to related systems, three design decisions stand out:

"Bottom" as a first-class citizen. In a lazy language, undefinedness is an ever-present concern, in that variables are not necessarily bound to well-defined values. For this reason, MProver's default mode of reasoning treats the undefined value (that is, the value of diverging or erroring computations) as a "first-class citizen." For example, proofs by case analysis must consider as one case the possibility that the expression being analyzed is undefined. By the same token, properties like "f is strict in its third argument" can be expressed directly in the logic.

More general notion of equality for potentially infinite structures. Systems like Coq have a rather restrictive notion of equality for coinductive types. Equality in Coq is intensional: that is, two expressions of the same type are equal if and only if they evaluate to the same normal form. Coq will generally refuse to β-reduce applications of functions producing a coinductive type, since this may result in nontermination and thus compromise logical soundness. Thus when working with coinductive types, one generally must define a weaker notion of *bisimulation* in lieu of equality *per se*. Because MProver separates the universes of programs and proofs—program terms are only an *object* of logical reasoning and are not themselves treated as logical proofs—it is possible to define a notion of equality over infinite structures that is much easier to deal with.

Type classes to structure verification arguments. Haskell's type class system enables programs to be written in terms of signatures, rather than particuar type structures. Perhaps the most common example is that of monads, which can be used to model a wide variety of "notions of computation", such as I/O, mutable state, and nondeterminism. Programs written to target the *Monad* type class can then be reused in any computational setting, and new computational settings may be added to the *Monad* class as long as operations for sequencing (*bind*) and for injection of effect-free computations (*return*) are defined.

In general, a type class is associated not just with a set of type signatures, but also with an implicit *specification* or *contract* governing how the operations are supposed to behave; for example, *Monad* instances are, very loosely stated, supposed to have the properties that sequential composition of computations is associative and that *return* is a left- and right-unit with respect to sequential composition. Haskell's type system does not check these properties; indeed, it cannot even express them directly. MProver makes the contract explicit; it augments the *Monad* class by adding *proof obligations* for the monad laws, as pictured in Figure 1. This approach allows not just programs, but also proofs, to be parameterized over all monadic notions of computation, enabling a modular style of proving that closely parallels the familiar vocabulary of functional programming idioms.

```
class Monad m where
  (>>=)      :: m a → (a → m b) → m b
  return     :: a → m a
  leftunit   ::: ∀ (x :: a) (f :: a → m b), return x >>= f = f x
  rightunit  ::: ∀ (x :: m a), x >>= return = x
  assoc      ::: ∀ (x :: m a) (f :: a → m b) (g :: b → m c),
                 (x >>= f) >>= g = x >>= λy → f y >>= g


instance Monad Maybe where
  (Just x) >>= f  = f x
  Nothing  >>= _  = Nothing
  return          = Just
  leftunit        = leftunitMaybe
  rightunit       = rightunitMaybe
  assoc           = assocMaybe


leftunitMaybe  = Foralli (x :: a) (f :: a → m b),
  join ::: return x >>= f = f x

rightunitMaybe = Foralli (x :: m a), case x of
  undefined → join ::: undefined >>= return = undefined
  Nothing   → join ::: Nothing >>= return = Nothing
  Just v    → join ::: Just v >>= return = Just v
assocMaybe     = Foralli (x :: m a) (f :: a → m b) (g :: b → m c), case x of
  undefined → join ::: (undefined >>= f) >>= g = undefined >>= (λy → f y >>= g)
  Nothing   → join ::: (Nothing >>= f) >>= g = Nothing >>= (λy → f y >>= g)
  Just v    → join ::: (Just v >>= f) >>= g = Just v >>= (λy → f y >>= g)
```

Fig. 1: The *Monad* type class in MProver, and an example instance

The remainder of this paper proceeds as follows. Section 2 gives a definition of the core language of programmatic expressions, logical formulas, proof terms, and tactics, and concludes with a few simple examples. Section 3 sketches the extension of the language of Section 2 with type classes, illustrated by a monadic equational reasoning proof adapted from Gibbons and Hinze [5]. Section 4 discusses related work, and Section 5 concludes.

2 Expressions, Formulas, Proofs, and Tactics in MProver

In this section we describe the basic design of the MProver language. We shall defer discussion of the type class system to Section 3. The language may be divided into two parts: the *programming* fragment and the *proving* fragment. The programming fragment is a pure, lazy functional language with type classes—essentially a subset of Haskell 98 [16]. In the proving fragment, proofs are expressed as terms in a λ-calculus-like language, but this language is entirely distinct from the language of programs and is not intended for evaluation other than possibly for metatheoretic purposes.

An MProver script consists of one or more top-level declarations. A declaration may be a datatype declaration, a program term declaration, or a theorem declaration. The syntax for datatype and program term declarations is identical to (a subset of) Haskell, while theorem declarations have the form:

var ::: *formula*
var = *proof*

Here the *var* is the name of the theorem, the *formula* is the definition of the theorem, and the *proof* is a proof of the theorem. The symbol ::: should be read as "proves the formula", by analogy with the Haskell symbol :: which is read "has type". The languages of formulas and proofs, and the relationship between them, are discussed in Section 2.3.

2.1 Expressions

As a matter of terminology, we will refer to programmatic terms as *expressions* and proof terms as *proofs*. We will consider a subset of the full expression language; the grammar of this subset is given in Figure 2. The full language is more fully featured than this, but the simpler formulation of Figure 2 will suffice for our discussion of the essential characteristics of the language.

expr ::=	λ *var* $\rightarrow$ *expr*	(abstraction)
|	*expr expr*	(application)
|	*var*	(variables)
|	*ctor*	(data constructors)
|	**let** (*var* = *expr*)* **in** *expr*	(let-binding)
|	**case** *expr* **of** (*pat* $\rightarrow$ *expr*)+	(case)
|	**undefined**	(undefined value)
pat ::=	*var*	(variables)
|	*ctor pat**	(constructed values)
|	_	(wildcard)

Fig. 2: Grammar for MProver Expressions

The semantics of MProver expressions is a standard non-strict semantics. As in Haskell, **let** bindings are recursive.

2.2 Formulas and Proofs

The main design goal of MProver is to support equational reasoning. For this reason, the language of program properties—called *formulas*—comprises just statements of (in)equality between program terms, logical implication, and universal quantification over expressions. The grammar of formulas is given in Figure 3. Note that the $\forall$ symbol in formulas is unrelated to the `forall` keyword used for higher-rank polymorphism in Haskell.

Figure 4 gives the grammar of proof terms. In order: the **Foralli** and **Assume** forms are used to introduce variables ranging over expressions and proofs of formulas, respectively. These variables may be referenced inside of a proof;

formula ::=	∀ (*var* :: *type*), *formula*	(universal quantification)
|	*formula* → *formula*	(implication)
|	*expr* = *expr*	(equality)
|	*expr* ≠ *expr*	(inequality)

Fig. 3: Grammar for MProver Formulas

universally quantified proof terms may be instantiated by expressions; implication proofs may be applied to proofs of the antecedent (modus ponens); proof expressions may be annotated with formulas as a hint to the proof checker; and **case**-proofs allow casewise splitting on program terms. Of the remaining proof forms, **eval**, **clash**, **refl**, **trans**, and **symm** may be viewed as directly reflecting equational judgments that follow from the reduction semantics. The **cong** form is used for congruence proofs (that is, substituting equals for equals).

proof ::=	**Foralli** (*var* :: *type*), *proof*	(forall-introduction)
|	**Assume** (*var* ::: *formula*), *proof*	(assumption)
|	*var*	(variables)
|	*proof expr*	(application 1)
|	*proof proof*	(application 2)
|	*proof* ::: *formula*	(annotation)
|	**case** *expr* **of** ($pat_\perp$ → *proof*)*	(pattern matching)
|	**eval**	(evaluation)
|	**clash**	(constructor-clash)
|	**cong** *proof*	(congruence)
|	**refl**	(reflexivity)
|	**trans** *proof proof*	(transitivity)
|	**symm** *proof*	(symmetry)
$pat_\perp$ ::=	*var*	(variables)
|	*ctor* $pat_\perp$*	(constructed values)
|	_	(wildcard)
|	**undefined**	(bottom)

Fig. 4: Grammar for MProver Proofs

2.3 Classification Rules

The process of proof checking, that is determining that a proof proves a formula, is called *classification*, and is akin to type checking. The classification rules are given in Figure 5. Contexts Γ contain assumptions of three forms: (1) $x ::: \phi$, indicating that variable x ranges over proofs of formula ϕ; (2) $x :: t$, indicating that variable x ranges over expressions of type t; and (3) $x = e$, indicating that variable x is bound to expression e. The "initial" Γ used by the proof checker

introduces all assumptions of the third type by binding top-level symbols to their definitions; there is no let-binding construct for proof terms.

The rules in the left column are essentially standard rules for assumption, abstraction, and application. The rule for **case** proofs has a couple of important features: first, pattern matching must be exhaustive, and the **undefined** case must be considered. Second, the formula ϕ that is proven inside the case alternatives is parameterized by the different cases: for example, if a proof of the formula $\forall\,(b :: \mathit{Bool}), \mathit{not}\,(\mathit{not}\,b) = b$ is done by case analysis of b, the body of the respective case alternatives must prove $\mathit{not}\,(\mathit{not}\,\mathit{True}) = \mathit{True}$, $\mathit{not}\,(\mathit{not}\,\mathit{False}) = \mathit{False}$, and $\mathit{not}\,(\mathit{not}\,\mathbf{undefined}) = \mathbf{undefined}$. We assume, for the sake of brevity, that all patterns are either a constructor applied to variable patterns, **undefined**, or the wildcard pattern (i.e. that patterns are not nested).

With respect to **cong**, note that due to MProver's lazy semantics, this one rule enables us to do all kinds of substitutions. For example, if we know that

$$\frac{(x ::: \phi) \in \Gamma}{\Gamma \vdash x ::: \phi} \qquad \frac{e_1 \rightsquigarrow_\Gamma e_2}{\Gamma \vdash \mathbf{eval} ::: e_1 = e_2}$$

$$\frac{\Gamma, (x :: t) \vdash p ::: \phi}{\Gamma \vdash \mathbf{Foralli}(x :: t),\ p ::: \forall (x :: t), \phi} \qquad \frac{C \neq C'}{\Gamma \vdash \mathbf{clash} ::: C\ e_1 \cdots\ e_n \neq C'\ e'_1 \cdots\ e'_m}$$

$$\frac{\Gamma, (x ::: \phi) \vdash p ::: \phi'}{\Gamma \vdash \mathbf{Assume}(x ::: \phi),\ p ::: \phi \to \phi'} \qquad \frac{\Gamma \vdash p ::: e' = e''}{\Gamma \vdash \mathbf{cong}\ p ::: e\ e' = e\ e''}$$

$$\frac{}{\Gamma \vdash \mathbf{refl} ::: e = e}$$

$$\frac{\Gamma \vdash p ::: \forall (x :: t), \phi \quad \Gamma \vdash e :: t}{\Gamma \vdash p\ e ::: [x/e]\phi} \qquad \frac{\Gamma \vdash p_1 ::: e_1 = e_2 \quad \Gamma \vdash p_2 ::: e_2 = e_3}{\Gamma \vdash \mathbf{trans}\, p_1\, p_2 ::: e_1 = e_3}$$

$$\frac{\Gamma \vdash p ::: e_1 = e_2}{\Gamma \vdash \mathbf{symm}\, p ::: e_2 = e_1}$$

$$\frac{\Gamma \vdash p ::: \phi \to \phi' \quad \Gamma \vdash p' ::: \phi}{\Gamma \vdash p\ p' ::: \phi'}$$

$$\frac{\begin{array}{c}\Gamma, x_{11} :: p_{11} \cdots x_{1m_1} :: t_{1m_1} \vdash p_1 ::: [x/(C_1\ x_{11}\ \cdots\ x_{1m_1})]\phi \\ \vdots \\ \Gamma, x_{n1} :: t_{n1} \cdots x_{nm_n} :: t_{nm_n} \vdash p_n ::: [x/(C_n\ x_{n1}\ \cdots\ x_{nm_n}]\phi \\ \Gamma \vdash p_\perp ::: [x/\mathbf{undefined}]\phi\end{array}}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \{(C_1\ x_{11}\ \cdots\ x_{1m_1}) \to p_1; \cdots; (C_n\ x_{n1}\ \cdots\ x_{nm_n}) \to p_n; \mathbf{undefined} \to p_\perp\} ::: [x/e]\phi}$$

(Note: The **case** rule assumes that $C_1 \cdots C_n$ are all the constructors for some data type, that $C_1 \cdots C_n$ have arities $m_1 \cdots m_n$ respectively, that no x_{ij} is free in ϕ, and that x is not the same variable any x_{ij}.)

Fig. 5: Proof Classification Rules

$f = g$ and we want to use this to prove that $mapfxs = mapgxs$, we can just make up a new λ-abstraction, with a fresh variable representing the substitution sites, and use **cong** apply that to our proof that $f = g$. A few **eval** steps will then give us the substitution we desire. The price we pay for this parsimony is that it is a bit tedious if such proofs done by hand; however, the **subst** tactic described below automates this process.

2.4 Recursive Data Structures

The presence of recursive data structures necessitates some kind of support for coinduction. For this, we turn to Coq for inspiration. As an example, consider the type of lazy lists: Figure 6 gives two definitions. The first is the definition of the lazy list data type; the `CoInductive` keyword indicates that `LLists` may possibly be infinite. The second definition is of a *bisimulation* – that is, a coinductively defined predicate that is weaker (i.e. identifies strictly more expressions) than Coq's definitional equality.

```
CoInductive LList (A:Type) : Type :=
| LCons : A -> LList A -> LList A
| LNil  : LList A.

CoInductive bisim (A:Type) : LList A -> LList A -> Prop :=
| bisim_LCons : forall (x:A) (l1 l2:LList A),
    bisim l1 l2 -> bisim (LCons x l1) (LCons x l2)
| bisim_LNil  :
    bisim (LNil A) (LNil A).
```

Fig. 6: Lazy lists and bisimilarity in Coq

To support reasoning about potentially infinite codata in MProver, each constructor in a data-declaration induces an analogous constructor form for equality proofs. In the example of Figure 7, the equality-proof forms for lazy lists are given; notice that they have essentially the same form as the constructors for the Coq bisimulation. In MProver, we simply overload the symbols *Nil* and *Cons* as constructors for a coinductively-defined bisimulation on lazy lists. This is the same bisimulation that we used in the Coq example: intuitively, if we have a proof p that expressions e and e' are equal, and a proof p_l that lists l and l' are equal, then just as we can "cons" e and e' onto l and l', so too can we cons p onto p_l, thus proving $e : l = e' : l'$.

Guardedness. Coinductive proofs, then, are constructed as corecursive proof terms—that is, they are defined in terms of themselves. As always, this raises a red flag: unrestricted use of recursion would render MProver's logic unsound, allowing us to prove any theorem by appeal to itself. In order to mantain soundness, we require that all recursive proof applications use *guarded* recursion. Here

data $List\ a\ =\ Nil \mid Cons\ a\ (List\ a)$

$$\frac{}{\Gamma \vdash Nil ::: Nil = Nil} \qquad \frac{\Gamma \vdash p ::: x = x' \quad \Gamma \vdash ps ::: xs = xs'}{\Gamma \vdash Cons\ p\ ps ::: Cons\ x\ xs = Cons\ x'\ xs'}$$

Fig. 7: Lazy lists and their coinduction rules in MProver

again, *guardedness* is the same guardedness condition used by Coq [7]. In a nutshell, this means that any recursive application of a proof term must occur inside an equality constructor application. The proof of Figure 8, to be discussed in greater detail shortly, illustrates this: the recursive application of `mapfusion` is an immediate argument to `Cons`, and thus the recursion is guarded.

2.5 Tactics and Syntactic Sugar

The core proof language of Figure 4, while quite expressive, is a bit inconvenient when large numbers of evaluation steps must take place. Consider, for example, the simple property $id\ (id\ id) = (id\ id)\ id$, where id is a global symbol defined as $id = \lambda x \to x$. Probably the most obvious proof of this requires four **eval** steps to reduce both sides of the equation to $id\ id$, and an application of **symm** to link the two halves of the proof together. The result may not be intimidating to the proof checker, but from a human's point of view it is rather tedious and unpleasant:

```
trans (eval ::: id (id id) = (\ x -> x) (id id))
      (trans (eval ::: (\ x -> x) (id id) = id id)
             (symm (trans (eval ::: (id id) id = ((\ x -> x) id) id)
                          (eval ::: ((\ x -> x) id) id = id id))))
```

For this reason, we extend the core language with a tactic called **join**. Given left- and right-hand side expressions e_1 and e_2, the **join** tactic works by repeatedly applying reduction steps to e_1 and e_2 until one of the following happens:

- e_1 and e_2 reach e_1' and e_2', respectively, where e_1' and e_2' are α-equivalent. Then **join** succeeds.
- e_1 and e_2 reach $C_1\ e_{11} \cdots\ e_{1n}$ and $C_2\ e_{21} \cdots\ e_{2m}$ where C_1 and C_2 are different constructors. Then **join** fails.
- e_1 and e_2 reach $C_1\ e_{11} \cdots\ e_{1n}$ and $C_2\ e_{21} \cdots\ e_{2n}$ where some e_{1i} is not α-equivalent to e_{2i}. Then for each e_{1j} and e_{2j} such that e_{1j} and e_{2j} are not α-equivalent, recursively attempt to join e_{1j} with e_{2j}. If all the recursive calls succeed, then **join** succeeds. If any recursive call fails, then **join** fails.

This procedure, which happens during the proof-checking phase, produces a proof term like the one given above; this reduces the burden on the user, who would otherwise have to construct tedious step-by-step reduction proofs by hand.

At the same time, it does not complicate the underlying theory, since even if the tactic succeeds the generated proof term will still be checked according to the rules of Figure 2.3.

A second tactic, called **subst**, comes in handy when it is necessary to rewrite underneath constructors or inside λ-abstractions; given a proof p that $e_1 = e_2$, **subst** will construct a proof that $e = e'$ if e' can be obtained by substituting all occurrences of e_1 with e_2 in e. This tactic makes extensive use of the **cong** rule.

2.6 Syntactic Sugar for `trans`

The applicative syntax for constructing **trans**-proof terms is a little bit unwieldy in practice. This is unfortunate, considering that equational reasoning proofs, often quite transitivity-heavy, are exactly what MProver is meant for! To ameliorate this, we supply a little bit of syntactic sugar, vaguely inspired by Haskell's **do**-notation for monads. Any proof term of the form:

```
[ e1 = e2 { p1 } ... = en { pn-1 } ]
```

will be desugared to:

```
(trans (trans (trans ... (p1 ::: {e1 = e2}) (p2 ::: {e2 = e3}))
                     ... (pn-1 ::: {en-1 = en})))
```

The result, illustrated in Figure 8 and in Figure 10, bears a reassuringly close resemblance to a "textbook" equational reasoning proof.

2.7 Example: Map Fusion

Having built up the requisite machinery, we can now present a more involved example of an MProver proof, given in Figure 8. The proof is of the familiar map fusion property, over the lazy list type defined previously in Figure 7. Assume that `map` and the function composition operator `.` are defined in the standard way. The proof then breaks down into three cases: one where the input list is undefined, one where it is empty, and one where it is a cons cell. In the first two cases, the desired property follows simply from evaluation (`join`). The third case is slightly more complicated, requiring the use of coinduction. Here the first use of (`join`) pulls the `Cons` constructor out front, and combines the applications of `g` and `f` with function composition. We then appeal to the coinduction rule for `Cons` to rewrite the tail of the list into the desired form. In the final step, simple evaluation gives us the result we want.

3 Type Classes

Let us now turn our attention to MProver's extended notion of type classes. Type classes were introduced in Haskell to allow for ad-hoc polymorphism—that is, overloading of functions and operators—in a natural, extensible way. Viewed

```
mapfusion ::: Forall (f::a -> b) (g::b -> c) (l::List a),
  map g (map f l) = map (g . f) l
mapfusion = Foralli (f::a -> b) (g::b -> c) (l::List a),
  case l of
    undefined -> join ::: map g (map f undefined) = map (g . f) undefined
    Nil       -> join ::: map g (map f []) = map (g . f) []
    Cons x xs ->
      [ map g (map f (Cons x xs))
      = Cons ((g . f) x) (map g (map f xs)) { join }
      = Cons ((g . f) x) (map (g . f) xs)   { Cons refl (mapfusion f g xs) }
      = map (g . f) (Cons x xs)             { join }
      ]
```

Fig. 8: Map fusion

another way, type classes can be seen as supporting *modularity* and *abstraction* through well-defined interfaces: the programmer may declare a type to be an instance of any type class simply by supplying a set of functions or operators with the right type signature; programs whose types are parameterized over members of that class can then be reused on new instances.

The Haskell community has developed a rich vocabulary of functional programming abstractions grounded in abstract algebra and category theory, and type classes are the language in which these abstractions are expressed [26]. However, a type class is often associated not just with a set of type signatures, but also with one or more *laws*. For example, any instance type a of the class *Monoid* must be associated with operators $mempty :: a$ and $mappend :: a \to a \to a$. [3] This requirement is enforced by Haskell's type system. It is *also* expected, however, that the operators follow certain laws: namely that *mempty* is a left and right identity with respect to *mappend*, and *mappend* is associative. This requirement is not checked mechanically by, nor even expressible in, Haskell's type system.

MProver supports richer specifications by extending type classes with associated formulas expressing the laws that instances of the class must obey. Any declared instance must contain not only definitions for the operators, but also proofs that the operators follow the laws. Figure 1 illustrates a well-known example of a type class in Haskell, that of *monads*[4], extended with proof obligations for the monad laws, along with a particular instance of that class (the *Maybe* type familiar to Haskell programmers).

[3] There is also a third operator $mconcat :: [a] \to a$, which has a default implementation that may be overridden if desired for reasons of efficiency.

[4] We omit *fail* from our definition since it is not really part of the mathematical notion of a monad, though its inclusion would cause no problems.

3.1 Monadic Equational Reasoning

Our approach to monadic equational reasoning shares much with that taken by Gibbons and Hinze [5]. In particular, we program and prove in terms of interfaces that axiomatize the behaviors of particular monadic effects, rather than constructing a particular implementation of that interface (though this can certainly be done). Figure 9 contains an MProver definition of Gibbons and Hinze's *MonadFail* and *MonadExcept* classes: that is, subclasses of *Monad* supporting the throwing and handling of exceptions. In Figure 10, we use this to prove the purity (meaning no uncaught exceptions) of a "fast product" function which takes the product of a list, but first scans the list, throwing an exception if a zero is found; this exception will be caught, and the function will return zero in this case.

As it happens, there are a few stipulations having to do with definedness that are not emphasized that strongly in the cited work, but must be made explicit here (though proofs are omitted for space reasons). In particular, we require that scanning the list for zero—that is, evaluating the expression `0 ‘elem‘ xs`—will terminate. Second, we must stipulate that the product function itself is short-circuiting: the obvious definition `foldr (*) 1` will not work, because it is possible that this will diverge even when `0 ‘elem‘ xs = True`. These sorts of stipulations seem to come up sufficiently often in real-world equational proofs that it might well make sense to extend the logic to make them explicit; this is discussed further in Section 5.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadFail}\ m\ \textbf{where} \\
\quad \mathit{fail} \quad :: \ m\ a \\
\quad \mathit{failLeftZero} ::: \forall\,(x :: m\ a),\ \mathit{fail} \gg x = \mathit{fail} \\
\textbf{class}\ \mathit{MonadFail}\ m \Rightarrow \mathit{MonadExcept}\ m\ \textbf{where} \\
\quad \mathit{catch} \quad :: \ m\ a \to m\ a \to m\ a \\
\quad \mathit{exceptLeftUnit} \quad ::: \forall\,(x :: m\ a),\ \mathit{catch}\ \mathit{fail}\ x = x \\
\quad \mathit{exceptRightUnit} ::: \forall\,(x :: m\ a),\ \mathit{catch}\ x\ \mathit{fail} = x \\
\quad \mathit{exceptAssoc} \quad ::: \forall\,(x\ y\ z :: m\ a),\ \mathit{catch}\ x\ (\mathit{catch}\ y\ z) = \mathit{catch}\ (\mathit{catch}\ x\ y)\ z \\
\quad \mathit{exceptPure} \quad ::: \forall\,(x :: a)\ (y :: m\ a),\ \mathit{catch}\ (\mathit{return}\ x)\ y = \mathit{return}\ x
\end{array}
$$

Fig. 9: The *MonadFail* and *MonadExcept* classes

4 Related Work

A major antecedent of this work is the Operational Type Theory implemented in the Guru programming language [20]. The most salient feature of Operational Type Theory for our purposes is its treatment of undefined computations: because OpTT directly encodes the (finite) sequences of reductions that are

```
ifredundant ::: Forall (b::Bool) (e::a),
                Assuming b =/= undefined,
                  if b e e = e
productspec ::: Forall (xs::[Int]),
                Assuming 0 `elem` xs = True,
                  0 = product xs
condlift ::: MonadExcept m =>
             Forall (b::Bool) (m1 m2 m3::m a),
             Assuming b =/= undefined,
                 catch (if b then m1 else m2) m3
               = if b then catch m1 m3 else catch m2 m3

fastprodpure ::: Forall xs::[Int],
                 Assuming 0 `elem` xs =/= undefined,
                   fastprod xs = return (product xs)
fastprodpure =
 Foralli xs::[Int],
 Assume (zerodecidable:::0 `elem` xs =/= undefined),
  [ fastprod xs
  = catch (work xs) (return 0)    { join }
  = catch
      (if 0 `elem` xs
         then fail
         else return (product xs))
      (return 0)                  { subst by (workspec xs) }
  = if 0 `elem` xs
      then catch fail (return 0)
      else catch
             (return (product xs))
             (return 0)           { condlift
                                      (0 `elem` xs) fail
                                      (return (product xs)) (return 0)
                                      zerodecidable }
  = if 0 `elem` xs
      then return 0
      else catch
            (return (product xs))
            (return 0)            { subst by (exceptLeftUnit (return 0)) }
  = if 0 `elem` xs
      then return 0
      else return (product xs)    { subst by (exceptPure (product xs)) }
  = if 0 `elem` xs
      then return (product xs)
      else return (product xs)    { subst by productspec }
  = return (product xs)           { ifredundant
                                      (0 `elem` xs)
                                      (product xs)
                                      zerodecidable }
  ]
```

Fig. 10: MProver formalization of Gibbons and Hinze's "fast product" function

required to establish equivalence of terms, the presence of nonterminating computations does not compromise the soundness of its proving fragment. There is much work on dealing with non-termination and infinite structures in existing theorem proving systems. Coq [13] features support for coinductive types [7], encompassing both coinductive data structures and predicates. A similar design is used for coinduction in Agda. Neither Coq nor Agda, however, supports direct reasoning about undefined computations, as MProver and Guru do. Indirect approaches to dealing with undefined computations include the formalization of domain theory within Coq [1], and extensions to Coq's type theory [14, 15].

MProver is certainly not the first tool designed to support integrated development and verification of functional programs. Particularly closely related to MProver is the Sparkle prover for the Clean programming language [4]. Like MProver, Sparkle has first-class support for reasoning about undefined computations: just as **undefined** is essentially treated as a constructor for all data types in MProver, Sparkle introduces a special expression form, denoted $\perp$, for talking about undefined computations. Sparkle is built on a sophisticated system of tactics and hints, which often results an automatic or near-automatic proving process where typical properties of functional programs are concerned.

One difference between MProver and Sparkle lies in the semantic foundations. Reasoning in Sparkle takes place internally in a simplified version of the Clean language called Core Clean. MProver, by contrast, does not simplify programs to a core language. The semantics of Core Clean is based on lazy graph rewriting, whereas MProver uses what is essentially a call-by-name reduction semantics. Considerable work has also been done on using type classes in Sparkle [22], which should enable a type class-directed style of proving (one of "proving to specifications" rather than proving to structures) very similar to MProver. It appears, however, that Sparkle does not extend type classes with logical specifications like MProver does. Instead, it relies on a clever scheme of induction over the sets of defined instances; this allows proofs to leverage the semantic relationships among derived typeclass instances (e.g. if the *Eq* instance for type T is an equivalence relation, so too is the instance for type $[T]$). The advantage of this approach is that reasoning in Sparkle can be viewed as "external" to an even greater degree than in MProver; reasoning about type class-based programs is available even if the original program is completely innocent to logical specifications. Finally, it is worth noting that Sparkle has built-in support for explicit strictness [21]. This provides a very clean and reasonable alternative to the kind of "definedness stipulations" that permeate the proof of Figure 10. In Section 5 we will briefly sketch a similar approach to this problem that we are considering for MProver.

Type classes originated in Haskell [23] as a means of enabling ad-hoc polymorphism, and the idea has been reimplemented in Coq [19] and Isabelle/HOLCF [11]. The Haskell community has developed a number of theories and tools [8] for formal (and semi-formal) reasoning about Haskell programs. Recent work on static contract checking [24, 25] focuses on the automatic verification of pre- and postconditions for Haskell functions. SmallCheck [17] is a type-directed framework for automated testing of program properties, similar to QuickCheck [3]

but exhaustively testing all values of a type up to a certain "depth", rather than randomly generating test cases. Other tools include the Haskell Equational Reasoning Assistant [6], an AJAX-based tool for rewriting Haskell expressions while preserving semantic equality. The HasCASL project [18] has developed an extension of the algebraic specification language CASL with Haskell-like language constructs. Further investigations of logical aspects of Haskell-like languages, especially with regard to laziness, may be found in [9, 12].

5 Summary and Future Work

This paper has outlined the design of a proof-checking system for a lazy functional language, called MProver. The work described here is still at an early stage, but we expect that MProver will be a very useful tool for mechanized equational reasoning. As we develop our implementation further, we are interested in adapting more examples of pen-and-paper proofs already in the literature and mechanizing them with MProver.

A preliminary implementation of MProver is available from HackageDB, containing a few examples that may be of interest to the reader. As of this writing, we do not have a full development of MProver's metatheory. However, we believe that a soundness argument may be derived from Harrison and Kieburtz's semantics for Haskell, in a similar fashion to P-logic [9,12]. P-logic takes the view that if P is a predicate over (possibly undefined) values of a type T, then P *refines* T: that is, the denotation of P is a subset of the denotation of T. In adapting this to MProver, we may say that **Forall** formulas quantified over expressions of type T refine the type T. As it happens, the programming fragment of MProver as presented here does not contain such features as *seq*, $\sim$-patterns, and polymorphic recursion, which *are* present in P-logic and add significantly to the complexity of its semantics. Excluding these features means that the soundness argument for MProver will be considerably simpler; at the same time, we believe that building MProver's semantics on top of this work shows that MProver *could* be extended with these features without disturbing its semantic foundations.

We also expect that an operational semantics, and a corresponding soundness argument, can be derived from existing work on Operational Type Theory [20], if it is suitably adapted to handle lazy evaluation and infinite data structures; we intend to consider this in future work.

Extension: Termination Types. Haskell equational reasoning proofs very often make assumptions about the finiteness or definedness of an expression. There are two major reasons for this: (1) sometimes infinite or undefined values are a pathological case that we don't actually care about, and excluding them may simplify reasoning (permitting in particular the use of structural induction); and (2) sometimes the properties we want to prove for defined and/or finite values aren't actually true of undefined or infinite values. The design described here does not have an adequate mechanism for handling this.

A possible solution to this problem is to further augment MProver's type system with a system of *termination types*. In this system, termination annotations would refine types by attaching tags restricting types to the finite case, or to the defined case (or to both). A proof quantified over, say, finite lists, could then use structural induction (as opposed to *co*induction). Standard techniques for checking termination and productivity (e.g. structural/guarded recursion) could be integrated into the type checker. This idea bears a close similarity to Howard's work on pointed types [10] and the termination types of Trellys [2].

Acknowledgments. The authors would like to thank the anonymous reviewers of previous versions of this paper for their most insightful comments. Marko van Eekelen provided many helpful insights and pointers into the literature, especially relating to Sparkle's handling of type classes and (co-)induction.

References

1. Nick Benton, Andrew Kennedy, and Carsten Varming. Some Domain Theory and Denotational Semantics in Coq. In *TPHOLs '09*, pages 115–130.
2. Chris Casinghino, Harley D. Eades III, Garrin Kimmell, Vilhelm Sjöberg, Tim Sheard, Aaron Stump, and Stephanie Weirich. The preliminary design of the Trellys core language. Talk and discussion session at PLPV 2011.
3. Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP '00*, pages 268–279.
4. Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem Proving for Functional Programmers. In *IFL '02*, pages 271–283.
5. Jeremy Gibbons and Ralf Hinze. Just do it: Simple monadic equational reasoning. In *ICFP*, September 2011.
6. Andy Gill. Introducing the Haskell Equational Reasoning Assistant. In *Haskell 2006*, pages 108–109.
7. Carlos Eduardo Giménez. Un calcul de constructions infinies et son application a la verification de systemes communicants, 1996. Ph.D. thesis.
8. Thomas Hallgren. Haskell Tools from the Programatica Project. In *Haskell '03*, pages 103–106.
9. William L. Harrison and Richard B. Kieburtz. The Logic of Demand in Haskell. *J. Funct. Program.*, 15:837–891, November 2005.
10. Brian T. Howard. Inductive, coinductive, and pointed types. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 102–109, New York, NY, USA, 1996. ACM.
11. Brian Huffman, John Matthews, and Peter White. Axiomatic constructor classes in Isabelle/HOLCF. In *TPHOLs '05*, pages 147–162.
12. Richard B. Kieburtz. P-logic: property verification for Haskell programs, 2002.
13. The Coq development team. *The Coq Proof Assistant Reference Manual*. LogiCal Project, 2010. Version 8.3.
14. Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Hoare Type Theory, Polymorphism and Separation. *J. Funct. Program.*, 18(5-6):865–911, 2008.
15. Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP '08*.

16. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, the Revised Report*. Cambridge University Press, 2003.
17. Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: Automatic Exhaustive Testing for Small Values. In *Haskell '08*, pages 37–48.
18. Lutz Schröder and Till Mossakowski. HasCasl: Integrated higher-order specification and program development. *Theor. Comput. Sci.*, 410:1217–1260, March 2009.
19. Matthieu Sozeau and Nicolas Oury. First-class type classes. In *TPHOLs '08*, pages 278–293.
20. Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy Simpson. Verified Programming in Guru. In *PLPV '08*.
21. Marko van Eekelen and Maarten de Mol. Proof tool support for explicit strictness. In *IFL '05*, pages 37–54.
22. Ron van Kesteren, Marko van Eekelen, and Maarten de Mol. Proof support for general type classes. In *TFP '04*, pages 1–16.
23. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, pages 60–76.
24. Dana N. Xu. Extended static checking for Haskell. In *Haskell '06*, pages 48–59.
25. Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for Haskell. In *POPL '09*, pages 41–52.
26. Brent Yorgey. Typeclassopedia. Online at www.haskell.org/haskellwiki/Typeclassopedia, accessed May 31, 2012.