Elaborator Reflection: Extending Idris in Idris

David Christiansen
Indiana University
davidchr@indiana.edu

Edwin Brady
University of St Andrews
ecb10@st-andrews.ac.uk

Abstract

Many programming languages and proof assistants are defined by elaboration from a high-level language with a great deal of implicit information to a highly explicit core language. In many advanced languages, these elaboration facilities contain powerful tools for program construction, but these tools are rarely designed to be repurposed by users. We describe *elaborator reflection*, a paradigm for metaprogramming in which the elaboration machinery is made directly available to metaprograms, as well as a concrete realization of elaborator reflection in Idris, a functional language with full dependent types. We demonstrate the applicability of Idris's reflected elaboration framework to a number of realistic problems, we discuss the motivation for the specific features of its design, and we explore the broader meaning of elaborator reflection as it can relate to other languages.

1. Introduction

Writing programs that write programs, or *metaprogramming*, allows ordinary programmers to seize privileges that were once reserved for language implementers. This is especially true when metaprogrammers are able to extend the languages that they use to express themselves. Language extensions can eliminate the need for tedious boilerplate code, they can enable domain-specific idioms, notation and design patterns to be expressed directly, and they can enable optimizations that were not foreseen by the compiler's authors. In languages based on dependent type theory, metaprograms are widely used for automating the construction of proofs as well as programs.

Idris is a pure functional language with full dependent types. Idris is type checked in two stages: first, a metaprogram known as an *elaborator* translates the user-accessible language into a much smaller core language, known as TT, after which the TT program is type checked independently with a much simpler type checker.

Elaboration allows Idris to have high-level conveniences whose safety is guaranteed by the core language's type checker. These conveniences include implicit arguments that are found by the compiler, type classes[1], type-driven disambiguation of names, and local definitions. The core language has none of these. Elaborating each of these features affects the elaboration of the others. For example, resolving a type class might cause the value of an implicit argument to be discovered, which itself provides type information that enables a name to be disambiguated, which then might cause still more implicit arguments to be solved. This means that the elaboration procedures for each of these features must be able to share information and that they cannot be written in isolation.

A difficult metaprogramming problem calls for a powerful solution. Taking a cue from successful metaprogramming systems for proof automation, Brady (2013) based the Idris elaborator on the design of tactic-based interactive proof assistants, embedding proof tactics in a Haskell monad. The elaborator is then a function from high-level Idris abstract syntax trees to programs in this tactic language, which construct the desired TT program when run. The abstractions and effects in the tactic language, including holes to be filled in and unification problems yet to be resolved, enable the elaborators for each language feature to communicate without needing to know about each others' implementation details. Additionally, the tactic-based elaborator has proven to be robust in the face of extensions that were not conceived of at the time of its initial implementation, such as quasiquotation (Christiansen 2014). Unfortunately, the elaborator monad is part of the underlying language implementation, and modifications and extensions require changing the compiler, erecting a solid barrier between language users and language implementers.

In this paper, we introduce *elaborator reflection*, where Idris's elaboration framework is realized as a primitive monad in Idris itself. This empowers ordinary Idris users to write elaboration scripts in Idris, and running these elaboration scripts is as easy as type checking an Idris file, rather than rebuilding the compiler. With this new power at our fingertips, we discovered that Idris's elaborator monad turns out to be useful for more than just elaborating high level Idris programs. We extended Idris with the ability to write separate termination proofs for general recursive functions by automating the Bove-Capretta transformation (Bove and Capretta 2005), we automated the proofs of a number of the lemmas in Idris's standard library, and we demonstrated how to generate data types that match the schemas of external data sources based on Idris's type providers (Christiansen 2013), all without leaving the cozy confines of our favorite language.

Contributions

This paper makes the following contributions:

- We present *elaborator reflection*, a new paradigm for metaprogramming in typed functional languages, in which metaprograms output a core language and have access to the elaboration facilities of the high-level language's implementation.
- We demonstrate a concrete realization of elaborator reflection in the dependently typed functional language Idris.
- We demonstrate that Idris's elaborator reflection is general enough to support diverse metaprogramming tasks, including type class derivation, proof automation, interactive code generation, and even reimplementation of compiler features as libraries.

Additionally, elaborator reflection may provide an alternative route to self-hosting. We comment on this possibility in Section 7.

2. Motivating Examples

Elaborator reflection allows us to write metaprograms, describing mechanical procedures for automatically generating programs in

[1] Type classes are called "interfaces" in recent versions of Idris, but in the present paper we will use the Haskell terminology.

Idris's core language. In this section we briefly describe two examples which will benefit from elaborator reflection: automatic generation of termination predicates to help reason about totality; and automation of a family of proofs. Here, we describe the problems; later, in Section 5, we will describe how to solve these problems and others using elaborator reflection.

Before discussing more realistic examples, it can be instructive to see a small example of elaborator reflection in action. Firstly, we construct a metaprogram that will implement a monomorphic identity function at some particular type:

```
mkId : Elab ()
mkId = do x <- gensym "x"
          attack
          intro x
          fill (Var x); solve
          solve  -- the attack
```

The type annotation `Elab ()` declares that `mkId` is a metaprogram that uses elaborator reflection. The use of `do`-notation desugars to the monadic bind operator, just as in Haskell. The first step in generating the function is to construct a unique name for its argument using `gensym`, which returns a fresh name. The next tactic, `attack`, is used to delimit fresh scopes. It is described in detail in Section 4.1. The lambda term is constructed using `intro`, which automatically populates the type annotation of the lambda from the expected function type. The body of the lambda is filled with a reference to its bound variable `x`, after which the `solve` tactic is used to mark the body of the lambda and the scope introduced by `attack` as completed.

The metaprogram is invoked using the syntax `%runElab mkId`, which tells Idris to build a term by applying the elaborator `mkId`. It can be used in many different contexts that expect identity functions:

```
idNat : Nat -> Nat
idNat = %runElab mkId

idUnit : () -> ()
idUnit = %runElab mkId

idString : String -> String
idString = %runElab mkId

theAnswer : Int
theAnswer = (%runElab mkId) 42
```

We will return to a detailed description of elaborator reflection soon. First, we present two interesting metaprogramming problems.

2.1 Termination Predicates

Idris is primarily a programming language rather than a proof assistant. As such, it admits programs that it cannot determine to be total functions and programmers do not, in general, need to write termination proofs. To maintain its consistency as a logic, non-total programs are not reduced during type checking, but are otherwise compiled normally.

However, a user who has written a total function that does not pass the termination checker may later decide to use it in a type, either to prove some property about it or to be able to exploit its reduction behavior. Unfortunately, such a modification typically has a drastic negative impact on the readability of the function, commingling the administrivia of the termination argument with the interesting flow of the algorithm being implemented. Ideally, a user would specify an algorithm, and then receive a separate proof obligation for its termination condition, separating the mundane bookkeeping of the transformation from the interesting termination argument.

For example, a programmer might write a functional analogue of `quicksort`:

```
quicksort : List Nat -> List Nat
quicksort [] = []
quicksort (x::xs) =
  quicksort (filter (< x) xs) ++
  x :: quicksort (filter (>= x) xs)
```

Then, the programmer might decide that they want a total version of `quicksort`. Ideally, it should be sufficient to write:

```
make quicksort total by qsTotalityProof
```

where `qsTotalityProof` becomes the name of a proof obligation for a termination argument.

Rather than hard-coding such an interactive feature by directly modifying our language implementation, we can use the reflected elaborator to implement it as a library. Bove and Capretta (2005) demonstrated a means of rewriting a large class of functional programs that use general recursion into a form acceptable to proof assistants. The Bove-Capretta transformation of a program consists of an inductive family that represents the call graph of the program, indexed by the program's arguments, and a function that is structurally recursive over these call graphs. The inductive family is typically referred to as an *accessibility predicate*. For `quicksort`, the accessibility predicate is:

```
data QSAcc : List Nat -> Type where
    QSAccCase0 : QSAcc []
    QSAccCase1 : QSAcc (filter (< x) xs) ->
                 QSAcc (filter (>= x) xs) ->
                 QSAcc (x :: xs)
```

The well-foundedness of the accessibility predicate can serve as evidence that the program will return a value in finite time for a particular set of arguments that have been proven to be accessible, and it becomes possible to prove that the function is total by showing that all possible arguments are accessible. For `quicksort`, we write a helper function `qs'` by recursion over a `QSAcc` predicate:

```
qs' : (xs : List Nat) -> QSAcc xs -> List Nat
qs' [] QSAccCase0 = []
qs' (x :: xs) (QSAccCase1 l r) =
    qs' (filter (< x) xs) l ++
    x :: qs' (filter (>= x) xs) r
```

Additionally, accessibility predicates are collapsible (Brady et al. 2003) and can therefore be eliminated from the run-time behavior of the program, because the particular constructor of the accessibility predicate is always uniquely determined by the other arguments to the function. We will revisit this example in Section 5.1, and see how we can automatically generate accessibility predicates using elaborator reflection.

When the program to be translated makes use of nested recursion, the accessibility predicate will refer to the rewritten program, as the result of the nested call must be accessible. Thus, to support nested recursion, the method requires that the type theory support inductive-recursive definitions as set forth by Dybjer (2000). Both Idris and its reflected elaborator support inductive-recursive definitions.

2.2 Proof Automation

Proof obligations sometimes arise naturally in the course of dependently typed programming. Like Agda (Norell 2007), Idris is primarily designed to support the interactive, direct construction of proof terms in the same fashion as program terms.

When one cares about the structure of a proof object, rather than its mere existence, this approach is convenient. However, some proof objects are wholly uninteresting, and contribute only tedious details to a program text. In these cases, an automation system more like those of tactic-based proof assistants can be desirable, allowing these proofs to be produced with much less code.

As befits its roots in tactic-based interactive proof assistants, Idris's reflected elaborator can be used as a tactic language for proof automation. Here, we present a simple yet powerful proof automation procedure, named `mush` as an homage to the much more capable tactic `crush` in Chlipala (2011). We can apply `mush` as follows, for example, to prove the associativity of addition:

```
plusAssoc : (j, k, l : Nat) ->
            plus (plus j k) l = plus j (plus k l)
plusAssoc = %runElab mush
```

We can use `mush` to prove a variety of the properties that are proved by hand in Idris's Prelude, including that zero is a right identity of addition, that addition and subtraction cancel, that mapping a function over a list preserves the length of the list, that the length of two appended lists is the sum of the length of the input lists, and that the empty list is a right identity of appending lists. We will present the definition of `mush` in Section 5.2.

3. The Core Language and Elaboration

Idris's elaborator is written in Haskell in an elaboration monad that provides state and error handling effects. Full details of the elaborator, including the typing and evaluation rules for the core language, are given by Brady (2013). The typing rules are standard for a dependently typed core calculus. In this section we reprise the essential features for elaborator reflection. In particular, we show the syntax of the core language, TT, and its representation as high level Idris data types.

3.1 The Core Language, TT

Figure 1 shows the syntax of Idris's core expression language, TT. The purpose of elaboration is to translate a high level Idris program into a collection of data type definitions and pattern matching function definitions, built from TT expressions. Data Types are defined in the following form, where $\mathtt{T}$ is the *type constructor* and the $\mathtt{D}_i$ are the *data constructors*:

$$\underline{\text{data}} \quad \mathtt{T} : t \quad \underline{\text{where}} \quad \mathtt{D}_1 : t \mid \ldots \mid \mathtt{D}_n : t$$

All functions are defined by top-level pattern match clauses, in the following form:

$$\begin{array}{l} \mathtt{f} : t \\ \underline{\text{var}}\ \vec{x}_1 : \vec{t}_1 .\ \mathtt{f}\ \vec{t}_1 \ = \ t_1 \\ \ldots \\ \underline{\text{var}}\ \vec{x}_n : \vec{t}_n .\ \mathtt{f}\ \vec{t}_n \ = \ t_n \end{array}$$

Note that names which are used in the definition are explicitly bound as pattern variables using a $\underline{\text{var}}\ x : T$ binding. So, for example, the high level Idris program...

```
data Vect : Nat -> Type -> Type where
     Nil  : Vect Z a
     (::) : a -> Vect k a -> Vect (S k) a

vAdd : Num a => Vect n a -> Vect n a -> Vect n a
vAdd Nil       Nil       = Nil
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
```

...elaborates to the following definitions in TT:

$$\begin{array}{rll}
\text{Terms, } t ::= & c & \text{(constant)} \\
\mid & x & \text{(variable)} \\
\mid & b.\ t & \text{(binding)} \\
\mid & t\ t & \text{(application)} \\
\mid & \mathtt{T} & \text{(type constructor)} \\
\mid & \mathtt{D} & \text{(data constructor)} \\
\\
\text{Binders, } b ::= & \lambda x : t & \text{(abstraction)} \\
\mid & \underline{\text{let}}\ x \mapsto t\ :\ t & \text{(let binding)} \\
\mid & \forall x : t & \text{(function space)} \\
\mid & \underline{\text{var}}\ x : t & \text{(pattern variable)} \\
\\
\text{Constants, } c ::= & \mathtt{Type}_i & \text{(type universes)} \\
\mid & i & \text{(integer literal)} \\
\mid & str & \text{(string literal)}
\end{array}$$

Figure 1. TT expression syntax

$$\begin{array}{l}
\underline{\text{data}} \quad \mathtt{Vect} : \mathtt{Nat} \to (a : \mathtt{Type}) \to \mathtt{Type} \quad \underline{\text{where}} \\
\qquad\quad \mathtt{Nil} : \mathtt{Vect}\ \mathtt{Z}\ a \\
\qquad \mid \ (::) : (k : \mathtt{Nat}) \to (x : a) \to \\
\qquad\qquad\quad (xs : \mathtt{Vect}\ k\ a) \to \mathtt{Vect}\ (\mathtt{S}\ k)\ a
\end{array}$$

$$\begin{array}{l}
\mathtt{vAdd} : (a : \mathtt{Type}) \to (n : \mathtt{Nat}) \to \mathtt{Num}\ a \to \\
\qquad \mathtt{Vect}\ n\ a \to \mathtt{Vect}\ n\ a \to \mathtt{Vect}\ n\ a \\
\underline{\text{var}}\ a : \mathtt{Type},\ c : \mathtt{Num}\ a. \\
\quad \mathtt{vAdd}\ a\ \mathtt{Z}\ c\ (\mathtt{Nil}\ a)\ (\mathtt{Nil}\ a) \ = \ \mathtt{Nil}\ a \\
\underline{\text{var}}\ a : \mathtt{Type},\ k : \mathtt{Nat},\ c : \mathtt{Num}\ a, \\
\quad x : a,\ xs : \mathtt{Vect}\ k\ a,\ y : a,\ ys : \mathtt{Vect}\ k\ a. \\
\quad \mathtt{vAdd}\ a\ (\mathtt{S}\ k)\ c\ ((::)\ a\ k\ x\ xs)\ ((::)\ a\ k\ y\ ys) \\
\qquad = \ ((::)\ a\ k\ ((+)\ c\ x\ y)\ (\mathtt{vAdd}\ a\ k\ c\ xs\ ys))
\end{array}$$

Note that all of the function and pattern variables have been made explicit in the types.

3.2 Elaboration

Elaboration of a high level Idris program into TT involves traversing the syntax of the high level program, incrementally constructing a TT term. During elaboration, the TT term may contain *holes*, which stand for parts of the program which have not yet been elaborated. Elaboration is *type-directed*, meaning that the elaborator always knows the type of the term it is constructing. The elaborator may also generate *unification constraints* between the type of the term it is constructing, and the required type. Such constraints may often need elaboration to make further progress before they can be resolved. There are, therefore, several pieces of state which need to be maintained during elaboration:

- a *goal type*, which is a TT term representing the type to be inhabited by the current elaboration procedure;
- an incrementally constructed *proof term*, which is a TT term that must inhabit the goal type;
- a *hole queue* containing the holes that remain in the proof term;
- a collection of open unification constraints;
- a supply of fresh names; and miscellaneous pieces of state to implement other internal operations.

We refer to these collectively as the *proof state*.

Instead of maintaining a separate metavariable context that tracks the unique names, scopes, expected types, and equality constraints related to the holes in the term, the Idris elaborator equips TT with two additional binding forms, using the already-existing

$$
\begin{array}{rll}
b ::= & \ldots & \\
\mid & ?x : t & \text{(hole binding)} \\
\mid & ?x \approx t : t & \text{(guess)}
\end{array}
$$

Figure 2. TT_{dev} extensions

scoping and equality mechanisms for terms and contexts to represent metavariables in-place. Equality constraints are directly represented by having the bound hole variable occur multiple times in its scope. This style was pioneered by McBride's (1999) Oleg proof assistant. We refer to this extended core language as the *development calculus* TT_{dev}. Figure 2 shows the extensions to the syntax of hole and guess binders in TT_{dev}.

In TT_{dev}, the term $?x\ :\ t\ .\ e$ is the term e with an additional bound variable x. The typing rules for hole binders assign the type t to x in its scope, demanding no further justification because further elaboration will be required to determine the actual term of type t to be inserted. Because we are working modulo Idris's definitional equality which includes β-conversion, directly substituting a term for the hole x might trigger a sequence of reductions that interferes with further processing that we intend to perform. Thus, instead of directly substituting a term for a hole variable, we first construct a temporary binder that resembles a let but is computationally inert. This placeholder binder is called a *guess*. If the term e' is inserted into the hole x above, the resulting guess is written $?x \approx e' : t.\ e$. Then, when concerns about the precise structure of the body of the guess binder are no longer relevant, the guess can be substituted for the variable that it binds.

During elaboration, the names of holes and guesses are unique, so that the hole queue can contain just the names. When the hole queue is non-empty, elaboration is said to be *focused* at the hole whose name is at the head of the queue. Many tactics implicitly modify the focused hole.

The Idris elaborator interprets each high-level Idris construct as a computation in an elaborator monad:

```
data Elab a -- abstract
```

Elaboration scripts are written using a collection of basic *tactics*, which are small, reusable building blocks for proof term construction. Some of the most commonly used tactics are:

- `claim`, which wraps the focused subterm with a new hole binding for a given name and type;
- `fill`, which converts a hole into a guess given an appropriately-typed term; and
- `solve`, which substitutes the focused guess in its scope and focuses on the next term.

We will describe these and other tactics in detail in Section 4.1. The Idris elaborator (and hence the `Elab` data type) is written in Haskell, so any features of the language are written as programs in the `Elab` monad in Haskell. The purpose of elaborator reflection is to expose the `Elab` monad in Idris programs, thus allowing Idris extensions to be written directly in Idris itself. To use this, we need an Idris representation of TT programs, and an Idris API for the elaborator monad.

3.3 Reflection Data Types

Idris's reflection API uses a number of data types for representing terms and definitions, as shown in Figure 3.

The core language is represented by two separate data types: `Raw` and `TT`. `Raw` is used to represent terms that are to be submitted to the type checker, while `TT` represents terms produced by the

```
-- Variable names
data TTName = ...

-- Constants
data Const = I Int | Str String | ...

-- Binders
data Binder : (tmTy : Type) -> Type where
    Lam : (ty : a) -> Binder a
    Pi : (ty, kind : a) -> Binder a
    Let : (ty, val : a) -> Binder a
    PVar : (ty : a) -> Binder a
    Hole : (ty : a) -> Binder a
    Guess : (ty, val : a) -> Binder a

-- Terms which have not yet been typechecked
data Raw = Var TTName
         | RBind TTName (Binder Raw) Raw
         | RApp Raw Raw
         | RType
         | RConstant Const

-- Well typed, de Bruijn indexed terms
data TT = P NameType TTName TT
        | V Int
        | Bind TTName (Binder TT) TT
        | App TT TT
        | TConst Const
        | TType TTUExp
```

Figure 3. Outline of the reflected TT data type

type checker. The differences between them are that `Raw` does not mention universe levels and it does not distinguish between locally and globally bound names, with a single constructor for variable references. `TT`, on the other hand, decorates the type of types with information about universe levels and it uses a locally nameless representation, with de Bruijn indices for bound variables. Additionally, `TT` contains local caches of the types of global variables, allowing `TT` terms to be understood in isolation. Both representations of the core language have a uniform representation of binders, implemented by the data type `Binder`.

Idris names are represented using the `TTName` data type. In addition to names that users might write, with or without namespace qualifiers, `TTName` includes constructors for machine-generated unique names as well as names that result from inner scopes, such as `where` blocks. Additionally, it includes a constructor that is reserved for names generated by user elaboration scripts so that they can define globally-unique top-level names for helper definitions.

While the majority of the data types involved in elaborator reflection are used to represent details about the core Idris language, a few of them mention features of high-level Idris. In particular, top-level type signatures are represented by a list of `FunArg` s followed by a return type, rather than by a dependent function type in the core theory:

```
data Plicity =
    Explicit | Implicit | Constraint

record FunArg where
    constructor MkFunArg
    name    : TTName
    type    : Raw
    plicity : Plicity
    erasure : Erasure
```

This allows reflection scripts to specify whether each argument is to be passed implicitly, explicitly, or using type class resolution, as well as whether the arguments are intended to be runtime-relevant. By looking up this specification in the global context, elaboration scripts can determine the intended calling conventions of Idris functions and attempt to infer implicit arguments as users would expect. Additionally, record types such as `TyDecl`, `DataDefn`, and `FunDefn` describe new type declarations, data declarations, and pattern-matching functions to be added to the global context.

Finally, reflected error messages are represented as lists of the type `ErrorReportPart`, which contains constructors for embedding strings, Idris terms, names, and even other error messages in an error. This allows error messages that result from metaprograms to be rendered using Idris's pretty printer and to take advantage of some of Idris's interactive features.

3.4 Quasiquotation

Working directly with the `Raw` and `TT` data types when writing metaprograms in Idris has many of the same disadvantages as programming directly in TT. Christiansen (2014) described how to use Idris's elaborator to translate quotations of high-level Idris terms into quotations of TT terms. The expression syntax:

```
`( e )
```

refers to a quotation of the expression `e` for which a type must be inferrable, while the syntax:

```
`( e : t )
```

represents a quotation of a term at some particular type (Idris does not have a native type annotation syntax for expressions, as it is not necessary elsewhere). Within quotations, antiquotations are prefixed with a tilde (`~`). Quoted terms must be typeable in the global context, as they can be defined in one part of the program and used in another, and antiquotations must sometimes be used to properly instantiate implicit arguments. Quasiquotations can be used in both pattern and expression contexts, with antiquotations in patterns containing patterns. Additionally, both the `TT` and `Raw` data types can be quoted, and Idris uses the expected type to determine which representation to produce.

Idris also has a quotation syntax for `TTName`. Because Idris supports pervasive type-disambiguated overloading, there is a distinction between names as they are written in a source file and the full, qualified version that is eventually discovered during elaboration. The syntax `` `{{n}} `` is an unresolved quotation of the name `n` — that is, it is a quotation of the name `n` precisely as written. The similar syntax `` `{n} `` is a resolved quotation, where the name `n` is treated as a reference to a unique name in scope, and expanded with namespace qualifiers prior to reification as a `TTName`. If `n` cannot be uniquely resolved, then `` `{n} `` will result in an error. Typically, unresolved names should be used for new things being defined by a metaprogram, while resolved names should be used for existing definitions that the metaprogram will consult.

4. Elaborator Reflection

An Idris program consists of a series of declarations, which can be data types, records, functions, type classes, and instances, among others. Elaborator reflection adds a new declaration form:

```
%runElab e
```

Here, `e` is an Idris expression. This will first elaborate `e` with the goal type `Elab ()`. If successful, the result is then run as an *elaboration script* by a built-in interpreter that assigns elaboration effects to a collection of primitive elaboration scripts. This can be understood by analogy to run-time side effects. Like Haskell, Idris has a family `IO` of computations that are assigned effectful semantics by the language runtime: a program with type `IO a` produces a value of type `a`, potentially having used side effects to do so. Analogously, a program in `Elab a` produces a value of type `a`, potentially having used side effects in the elaborator. While `IO` actions are run implicitly by being assigned the name `main`, actions in `Elab` are run explicitly using the `%runElab` syntax.

Similarly, elaborator reflection adds a new expression form:

```
%runElab e
```

This works in exactly the same way as the declaration form, except that the elaboration script is run in the context of expression elaboration.

4.1 Elaborator Operations

`Elab` is an instance of the `Functor`, `Applicative`, and `Monad` type classes. Additionally, a backtracking failure-handling operator is used to implement `Alternative`, allowing the standard Idris choice operator `<|>` to be used to express error recovery. Errors can occur when tactics are misused, or they can be explicitly signaled by the `fail` tactic, which takes a reflected error message as an argument.

The additional effects available in `Elab` largely correspond to the low-level tactics mentioned in Section 3.2, described in detail by Brady (2013). While Brady's article distinguishes between expression and definition elaborators, with the latter invoking the former, the reflected elaborator allows the mixing of both forms of operation and thus expands these tactics with operators for both querying and extending the global context. This is to allow the same elaboration code to be used in both expression and definition contexts, as well as to allow elaboration scripts to produce expressions that depend on auxiliary definitions or data types. In this section, we describe the most important primitives in detail.

As described in Section 3, the elaboration monad has a state that includes a proof term with holes in it, a queue containing these holes, and a collection of open unification constraints. The following primitive operators in `Elab` manipulate or query the current proof state:

```
gensym : (hint : String) -> Elab TTName
```

Produce a unique name, which is useful when establishing a new binder.

```
focus : (h : TTName) -> Elab ()
```

Move the focus to the hole `h`, bringing it to the front of the hole queue, or fail if the `h` does not exist.

```
unfocus : (h : TTName) -> Elab ()
```

Move the hole `h` to the end of the hole queue, or fail if the `h` does not exist.

```
fill : (e : Raw) -> Elab ()
```

Place a reified term `e` with type t_e in the focused hole $?h : t_h \,.\, b$, converting it to a guess $?h \approx e{:}t.\ b$, where t unifies t_e and t_h. This unification can result in the solution of further holes or the establishment of additional unsolved unification constraints.

```
solve : Elab ()
```

Substitute the focused guess throughout its scope, eliminating it and moving focus to the next element of the hole queue. Fail if the focus is not on a guess.

```
claim : (n : TTName) -> (t : Raw) -> Elab ()
```

Establish a new hole binding named `n` with type `t`, surrounding the current focus.

```
apply : (op : Raw) -> (argSpec : List Bool) ->
        Elab (List TTName)
```

Apply the operator `op`, establishing holes for its arguments based on `argSpec`, a list of Booleans whose length is equal to the number of arguments that the operator will be applied to. A hole is established for each argument, with the type determined by the type of the operator, with the appropriate references to earlier holes in cases where the operator has a dependent type. If the corresponding Boolean is `False`, the hole is marked as unable to be solved by unification, while if the corresponding Boolean is `True`, the hole is marked as suitable to be solved automatically. The names of the established holes are returned. This tactic is not part of Brady's 2013 idealized tactic languge. For a discussion of the motivations for including it as a primitive in `Elab`, please refer to the discussion of direct vs. indirect reflection in Section 6.

```
compute : Elab ()
```

Normalize the current goal type.

```
rewriteWith : (rule : Raw) -> Elab ()
```

Attempts to rewrite the current goal using the equality proof `rule`. This tactic invokes the underlying elaborator for Idris's `rewrite ... in ...` syntax.

For example, in a proof with goal `Nat`, the term `plus Z (S Z)` could be instantiated by the following `Elab` script:

```
do [x, y] <- apply `(plus) [False, False]
   solve
   focus x; fill `(Z); solve
   focus y; fill `(S Z); solve
```

Each form of binder b in Figures 1 and 2 has an introduction associated tactic. A precondition of these tactics is that the focused hole is of the form $?h : t\,.\,h$ — that is, that the body of its scope consists directly of a reference to the hole-bound variable. If a hole binder were of the form $?h : t_1 \to t_2\,.\,f\ h$ and a tactic such as `intro ``{{x}}` were applied, the result would be the term $?h : t_2\,.\,\lambda x : t_1.\ f\ h$. However, this would cause the application of f to be ill-typed, as it expects an argument of type $t_1 \to t_2$, not an argument of type t_2. The binding tactics are:

```
attack : Elab ()
```

Restructure the focused hole to make it suitable for binding by establishing a new hole inside of a guess. If the focus is at $?h : t_1 \to t_2\,.\,f\ h$, then invoking `attack` will result in the term $?h \approx (?h' : t_1 \to t_2\,.\,h') : t_1 \to t_2.\ f\ h$ with the focus on h'. Invocations of binding tactics should be bracketed by `attack` and `solve`, representing the new scopes. Following a binding tactic, the focus is on a hole in the immediate scope of the introduced binding, so repeated invocations of `attack` are not necessary for repeated binders.

```
intro : (n : TTName) -> Elab ()
```

Wrap the current hole, which must have a function type, in a lambda. The provided name is used for function's argument.

```
forall : (n : TTName) -> (t : Raw) -> Elab ()
```

Wrap the focused hole $?h$: `Type` $\,.\,e$ in a dependent function type, yielding (`n : t`) $\to ?h$: `Type` $\,.\,e$.

```
patbind : (v : TTName) -> Elab ()
```

Introduce a new pattern binding var `v` around the hole, similarly to `intro` and `forall`.

```
letbind : (n : TTName) ->
          (v : Raw) -> (t : Raw) -> Elab ()
```

Introduce a new `let` binding around the focused hole, given a name `n` to bind, a term `v`, and a type annotation `t`. It is an error if `v` does not have type `t` in the focused context.

The `TT` equivalent of the Idris term `\t : Type => t -> Nat` can be built using the following script:

```
do attack
   intro `{{t}}
   arg <- gensym "arg"
   forall arg (Var `{{t}})
   fill `(Nat); solve
   solve -- the attack
```

The following queries against the global environment are provided as primitives in `Elab`:

```
getEnv : Elab (List (TTName, Binder TT))
```

Look up the lexical scope at the focused hole, or fail if there are no holes.

```
lookupTy : (n : TTName) ->
           Elab (List (TTName, NameType, TT))
```

Look up the core-language types of every overloading of a name `n` in the global context. The `NameType` indicates whether the name is a user-defined constant, a data constructor, or a type constructor.

```
lookupDatatype : (n : TTName) ->
                 Elab (List Datatype)
```

Look up the reified definitions of all data types whose names are overloadings of `n`.

```
lookupFunDefn : (n : TTName) ->
                Elab (List (FunDefn TT))
```

Look up the reified definitions of all functions whose names are overloadings of the `n`.

```
lookupArgs : (n : TTName) ->
             Elab (List (TTName, List FunArg, Raw))
```

Look up the high-level Idris calling conventions (that is, which arguments are to be found implicitly) for all overloadings of `n`.

The following operators provide an interface to the type checker and evaluator:

```
check : (env : List (TTName, Binder TT)) ->
        (tm : Raw) -> Elab (TT, TT)
```

Invoke the type checker to produce a fully-annotated term, considered with respect to a particular lexical environment `env`.

```
normalise : (env : List (TTName, Binder TT)) ->
            (tm : TT) -> Elab TT
```

Invoke the evaluator to normalize `tm` relative to the global context and the lexical context `env`.

Elaboration scripts can extend the global context with new declarations and definitions. Rather than a single-step process to define a function or data type, a two step process is employed where the type of a function or type constructor is first declared, followed by another operation to define either its pattern-matching equations or constructors. Under this scheme, the newly defined elements can be required to pass the type checker, but it is still possible to define functions and data types that refer to one another. Additionally, this allows elaboration scripts to leave behind abstract definitions to be later filled in by users, such as the totality proof obligations from Section 2.1. The operators for extending the global context are:

```
declareType : TyDecl -> Elab ()
```

Add a function type declaration to the global context.

`defineFunction : FunDefn Raw -> Elab ()`

Use a sequence of pattern-matching clauses to define a function.

`declareDatatype : TyDecl -> Elab ()`

Add the declaration of the type constructor for an inductive family to the global context.

`defineDatatype : DataDefn -> Elab ()`

Define an inductive family by its list of constructors.

`metavar : TTName -> Elab ()`

Solve the current hole as if it were a top-level Idris named hole. Not to be confused with the holes in TT_{dev}, which are an implementation technique for elaborators, Idris's holes represent unfinished user programs. These holes can later be solved using all of Idris's standard interactive features. This tactic allows metaprograms to delegate to human intelligence when necessary.

As an example of tactic-driven definitions, the Idris data type definition:

```
data B = T | F
```

can be equivalently constructed with the following metaprogram:

```
%runElab
  (do declareDatatype $
        Declare `{{B}} [] `(Type)
      defineDatatype $
        DefineDatatype `{{B}}
         [ Constructor `{{T}} [] (Var `{{B}})
         , Constructor `{{F}} [] (Var `{{B}})
         ])
```

and the Idris type declaration:

```
append : Vect n a -> Vect m a -> Vect (n + m) a
```

can be equivalently created using the metaprogram:

```
imp : TTName -> Raw -> FunArg
imp n t = MkFunArg n t Implicit Erased

exp : TTName -> Raw -> FunArg
exp n t = MkFunArg n t Explicit NotErased

appendDecl : TyDecl
appendDecl =
  Declare
    `{{append}}
    [ imp `{{a}} `(Type)
    , imp `{{n}} `(Nat)
    , imp `{{m}} `(Nat)
    , exp `{{xs}}
          `(Vect ~(Var `{{n}}) ~(Var `{{a}}))
    , exp `{{ys}}
          `(Vect ~(Var `{{m}}) ~(Var `{{a}}))
    ]
    `(Vect (plus ~(Var `{{n}})
                 ~(Var `{{m}}))
           ~(Var `{{a}}))
%runElab (declareType appendDecl)
```

In addition to the specialized searching performed to resolve type class instance dictionaries, Idris also contains a general-purpose proof search that attempts to use the constructors of inductive types to solve a goal. Proof search is invoked during elaboration when solving for implicit arguments decorated with the `auto` modifier as well as interactively by users during program development. In the reflected elaborator, proof search can be applied to the focused goal:

`search' : Int -> List TTName -> Elab ()`

Attempt to solve the focused hole using proof search, taking a search depth and a list of additional names to use as hints.

Many of the primitives in the reflected elaborator require already-formed terms as arguments. Sometimes, it can be non-trivial to produce these terms. In particular, reflected function definitions represent the defining clauses of pattern-matching definitions as pairs of terms, and patterns can have complicated relationships that are imposed by dependent pattern matching. To aid in the production of terms, elaboration scripts can recursively invoke the reflected elaborator:

`runElab : Raw -> Elab () -> Elab (TT, TT)`

Use the provided elaboration script to solve a reflected goal. The action is run in a fresh proof state with the provided type as the goal, and the resulting term and its type are returned on success.

Finally, to assist in the development of non-trivial elaboration scripts, some debugging tools are provided:

`debug : Elab a`

Halts the elaborator as if a fatal error had occurred and provide a detailed dump of the elaborator state, including the current proof term and the contents of the hole queue. This can be used to diagnose errors in elaboration scripts.

`debugMessage : List ErrorReportPart -> Elab a`

Halt and show all of the information displayed by `debug`, along with a user-specified message. The message, like the one provided to `fail`, is rendered using Idris's pretty printer.

Idris's type classes are elaborated to record types, and instances are elaborated to either records of functions or to functions from other instances dictionaries to new instance dictionaries. The database of registered instances for a given type class is a mapping from type class names to collections of instance names, and instance resolution attempts to combine these instances to create an inhabitant of a particular goal type. `Elab` contains effects for querying the instance database as well as registering new definitions as instances.

Additional operations include tactics to replace a hole with a pattern variable, look up the name and type of the focused goal, get the hole queue as a list of names, and check terms for conversion. There are also a collection of operators that reveal information about where `%runElab` was invoked, such as the lexically declared namespace, the precise source location, and the precedence and associativity declared for some operator.

4.2 Staging Issues

Elaboration of an Idris program proceeds from top to bottom, so names must be declared before use. Idris additionally supports `mutual` blocks, within which all type declarations are elaborated prior to elaborating all definitions, allowing for mutually recursive definitions. Some expressions, such as `case` expressions, require additional top-level definitions to be produced after elaboration, but these helper definitions are not accessible to other parts of the program, which means that users can be blissfully unaware of this elaboration order.

This account of name and definition visibility becomes more challenging in the presence of elaborator reflection. Like Template Haskell, Idris's elaborator reflection introduces tricky staging issues. The interpreter for elaboration scripts needs to be able to apply the functions that are referred to in the elaboration script. If these functions have not yet been defined, it is not possible to run

the script. In other words, the argument position of the %runElab operator is not referentially transparent!

For example, the following programs are not equivalent:

```
goodHelper : Elab ()
goodHelper = do g <- goalType
                case g of
                  `(Nat) => exact `(Z)
                  _ => fail [TextPart "Not a Nat"]

good : Nat
good = %runElab goodHelper

bad : Nat
bad = %runElab
       (do g <- goalType
           case g of
             `(Nat) => exact `(Z)
             _ => fail [TextPart "Not a Nat"])
```

In the case of good, the elaboration script will result in the right-hand side being defined as Z. In the case of bad, however, the elaboration script will throw an error. This is because Idris's expression elaborator has not yet created the helper definition that implements the case expression when it is elaborating the script on the right-hand side of bad, and the script is executed prior to the step in which the case expression's implementation is added to the global context. Users of the reflected elaboration machinery must be aware of the details of Idris's elaboration process.

5. Using the Reflected Elaborator

A metaprogramming system is of no use if it cannot be used to write interesting metaprograms. In this section, we deliver on our promises from Section 2 and describe additional experiences with the reflected elaborator.

5.1 Termination Predicates

In Section 2.1 we showed the following general recursive definition of quicksort:

```
quicksort : List Nat -> List Nat
quicksort [] = []
quicksort (x::xs) =
  quicksort (filter (< x) xs) ++
  x :: quicksort (filter (>= x) xs)
```

Idris cannot automatically determine that this function terminates. We can, however, show that it terminates using an accessibility predicate. Using elaborator reflection, we can automatically generate accessibility predicates by directly adapting Bove and Capretta's (2005) to rewrite Idris programs that make use of general recursion into a pair of an accessibility predicate and a function defined by recursion over the predicate. We begin by declaring the type of the elaboration procedure:

```
bc : (inFun, outFun, accPred : TTName) -> Elab ()
bc = do
```

The procedure bc takes three arguments: inFun, which is the partial program to be rewritten, outFun, which is the name to use for the rewritten function, and accPred, which is the name to use for the accessibility predicate. In the do block, the first step is to check that the provided name uniquely identifies a function:

```
(inFun', Ref, inTy) <- lookupTyExact inFun
   | _ => fail [TextPart "Not a function"]
```

The -Exact variants of the lookup operations require that a single name match and do not disambiguate. In Idris's extended do notation, the vertical bar specifies alternative actions to be taken instead of the remainder of the do block, in case a pattern-matching do binding fails to match its main pattern. In this case, the program terminates with an error message.

Next, the code generator declares the new data type and the new function. The lookupArgsExact query discovers the calling convention of a function — that is, which of its arguments are to be discovered automatically through unification or through type class resolution. After that, a new inductive family is declared using declareDatatype, which establishes a the formation rule or type constructor in the global typing context.

```
(_, args, res) <- lookupArgsExact inFun'
declareDatatype $ Declare accPred args `(Type)
```

Now that the accessibility predicate has been declared, the new function, which uses it, must be declared. The new functions arguments are those of the previous function along with the accessibility predicate applied to the other arguments.

```
let accArg = MkFunArg `{{acc}}
                  (mkApp (Var accPred)
                         (map (Var . name) args))
                  Explicit
                  NotErased
```

In this code, Var is a term constructor for variable references, taking a name as an argument, and name projects the name field from FunArg. The new argument accArg is added to the signature for the rewritten function:

```
declareType $
  Declare outFun (args ++ [accArg]) res
```

We declare the rewritten function's type and the type constructor of the accessibility predicate first, because the constructors of the data type may need to refer to the function, and those will not be typeable if it does not yet exist.

Now that the typing rules for the rewritten function and the accessibility predicate have been added, it is time to compute the constructors of the data type and the pattern-matching equations for the rewritten definition. The input the the transformation is the list of original clauses:

```
DefineFun _ oldClauses <- lookupFunDefnExact inFun'
```

Because each equation is simultaneously translated into a new equation and a corresponding constructor of the accessibility predicate, the result of processing the clauses is unzipped into constructors and new pattern-matching equations. The processClauses function is a direct implementation of Bove and Capretta's algorithm.

```
(newClauses, ctors) <-
  unzip <$> processClauses Z oldClauses
```

Finally, the data type is defined by its constructors and the rewritten function is defined using the new pattern-matching equations.

```
defineDatatype $ DefineDatatype accPred ctors
defineFunction $ DefineFun outFun newClauses
```

We can invoke bc using the %runElab directive:

```
%runElab (bc `{quicksort} `{{qs'}}  `{{QSAcc}})
```

The resulting definitions are:

```
data QSAcc : List Nat -> Type where
  QSAccCase0 : QSAcc []
  QSAccCase1 : QSAcc (filter (< x) xs) ->
               QSAcc (filter (>= x) xs) ->
               QSAcc (x :: xs)
```

and

```
qs' : (xs : List Nat) -> QSAcc xs -> List Nat
qs' [] QSAccCase0 = []
qs' (x :: xs) (QSAccCase1 l r) =
  qs' (filter (< x) xs) l ++
  x :: qs' (filter (>= x) xs) r
```

It is now up to a user to show that `QSAcc xs` is inhabited for all lists `xs` by defining a helper such as the following:

```
qsAcc : (xs : List Nat) -> QSAcc xs
```

This particular formulation is convenient when there may be some additional preconditions for totality. In cases where the function is actually total, it may be convenient to instead generate an unfulfilled proof obligation that the entire domain is accessible as well as a wrapper with the same signature as the original function. The wrapper then uses the result of this proof to call the rewritten function. Because this mechanism did not require modifications to the Idris compiler, users are free to customize it however they want, while still being able to rely on the unmodified underlying type checker for correctness, and features such as custom syntax rules[2] can be used to provide a syntax akin to that in Section 2.1

5.2 Proof Automation with Tactics

Previous versions of Idris supported a special-purpose tactic language for writing proofs at an interactive shell, and user interfaces reminiscent of dedicated proof assistants such as Coq were constructed to facilitate interactive proving. However, the special-purpose tactic language had a number of drawbacks: its extreme simplicity meant that it was not possible to build abstractions or nontrivial derived tactics, it was impossible to reuse Idris code in tactics, and it introduced a large number of new keywords into the language grammar. Elaborator reflection has subsumed this previous tactic language, allowing Idris's ordinary abstractions and definition forms to be used to build a richer language of tactics. Idris additionally includes a library of derived tactics that is named *Pruviloj*. In addition to extended versions of the primitive tactics that do things like introducing multiple lambda bindings at once, Pruviloj includes features such as an induction tactic that generates its own eliminators for inductive families and tactics for automatically discharging proof goals with absurd hypotheses. The eliminator-generation feature of Pruviloj replaces a built-in feature of the Idris compiler, and we aim to implement more of Idris in itself in the future.

To demonstrate the utility of the derived tactics in Pruviloj for proof automation, we now present the definition of the `mush` tactic from Section 2.2. The tactic will perform the following steps when given a goal of the form `(x : t) -> P x`:

1. Bind `x` under a lambda, introducing it as a local assumption
2. Perform induction on `x`, generating elimination principles as necessary
3. In each proof goal introduced by the induction, apply a non-inductive solver called `auto`.

```
mush : Elab ()
mush =
  do attack
     x <- gensym "x"
     intro x
     try intros
     induction (Var x) `andThen` auto
     solve
```

[2] `http://docs.idris-lang.org/en/latest/tutorial/syntax.html#syntax-rules`

This `mush` tactic uses derived tactics and tacticals such as `intros`, which introduces as many times as possible with automatically-generated names; `try`, which applies its argument but does nothing in case of failure; `induction`, which generates and applies an elimination principle to the appropriate value and motive, returning a list of holes corresponding to the cases for each constructor; and `andThen`, which applies its first argument, expecting a list of hole names, then applies its second argument in each of the holes.

The helper tactic `auto` performs the following steps:

1. Normalize the goal
2. If the goal is a universal quantification, introduce all quantified variables as local assumptions
3. Attempt to rewrite the goal with all equality proofs that are assumptions from induction
4. Attempt to dispatch the potentially-rewritten goal using a local hypothesis, or Idris's built-in proof search if that fails

```
auto : Elab ()
auto =
  do compute
     attack
     try intros
     hs <- map fst <$> getEnv
     for_ hs $
       \ih => try (rewriteWith (Var ih))
     hypothesis <|> search
     solve
```

The only new Pruviloj tatic employed in the definition of `auto` is `hypothesis`, which attempts to solve the goal with each variable in the local scope, failing if none apply. The tactic `search` is defined in the Prelude as `search' 100 []`. All of the control structures, including the `do`-notation, the recovery operator `<|>`, the iteration operator `for_`, and the mapping operator `<$>`, are standard Idris operators that can be used with any instances of `Monad`, `Alternative`, `Foldable`, and `Functor` type classes, which are analogous to their Haskell definitions.

5.3 Deriving Type Class Instances

Haskell supports deriving of instances for a collection of built-in type classes, freeing users from the burden of writing boilerplate Boolean equality comparisons and conversions to `String`. Idris has no similar feature. However, using elaborator reflection, we have been able to implement deriving of type classes such as `Eq` and `Show`. While deriving of `Eq` and `Show` are implemented as traditional code generators, we have also implemented a tactic that will derive decision procedures for Idris's propositional equality for a large class of data types, given a type signature. As decidable equality typically requires a number of pattern-matching cases that is quadratic with respect to the number of constructors in the data type, this can save a great deal of code.

5.4 Interaction with Idris Type Providers

Idris's type providers (Christiansen 2013), inspired by F# (Syme et al. 2013), allow arbitrary side effects to be performed during type checking. This lets Idris programs refer to the world in which they are found. Type providers in Idris differ from F#'s type providers, along with analogous uses of IO in Template Haskell, in a very important way: instead of using IO to do code generation, with the generated code spliced into the program prior to further type checking, Idris type providers compute a value using IO which is then included directly in the elaborated program. In particular, an Idris type provider is a term with type `IO (Provider a)`, where `Provider` is an error monad isomorphic to `Either String`. The

elaborator for the declaration syntax:

$$\texttt{\%provide } (x : a) \texttt{ with } p$$

executes p as an IO action. If p returns a success, then the contents of the success are unpacked and used as the definition of x, while if p returns an error message, elaboration halts and displays the message to the user. Because they are ordinary IO actions returning an ordinary Idris data type, Idris type providers manage to stay within the abstractions of the language. Idris's full dependent types enable these provided values to be used as codes in the technique of generic programming with universes (Altenkirch and McBride 2003; Benke et al. 2003). A single computed value can be used to instantiate an entire library of code at one particular type from a predetermined part of Idris's type system.

However, generic programming with universes does not come for free. Types described with a universe encoding do not enjoy the conveniences of native data types, such as convenient notations for pattern matching, and techniques like pattern synonyms cannot easily be specialized to a particular returned code. The syntactic overhead is substantial and difficult to avoid. Additionally, while Idris's optimizer and partial evaluator (Brady and Hammond 2010) may be able to reduce or remove the cost of carrying around codes at run time, this cannot be relied upon in general, which may lead to unpredictable changes in performance. For these reasons, generic programming with universes is not applicable to all type providers, and code generation may sometimes be a better option.

By first using a type provider to discover information about the surrounding world and then an `Elab` script to generate code, the full expressiveness of an approach based on code generation becomes available, while still maintaining the possibility of using ordinary Idris `IO` actions for the effectful portions.

Here, we demonstrate the definition of a type provider combined with an `Elab` script that, together, generate a record type based on a list of field/type pairs in a text file. Similar metaprograms are used for purposes such as generating data types based on database schemas.

An Idris record type consists of a single-constructor data type along with a collection of getters and setters that are named according to a particular convention. Getters project a single field from an instance of the record type, while setters construct a new instance that is the same as an old one, except with one field replaced by a new value. In Idris, the syntax `record {f} r` projects a field from `r` using the getter named `f`, while `record {f=y} r` invokes a setter named `set_f` to construct a modified version of `r`. In other words, to generate a record type named `R` with fields $\texttt{f}_1 : \texttt{t}_1, \ldots, \texttt{f}_n : \texttt{t}_n$, we generate the following type declaration:

$$\texttt{data R : Type where}$$
$$\texttt{MkR : (f}_1 \texttt{ : t}_1\texttt{) -> } \cdots \texttt{ -> (f}_n \texttt{ : t}_n\texttt{) -> R}$$

We also generate getters and setters for each field $\texttt{f}_k$:

$$\texttt{f}_k \texttt{ : R -> t}_1$$
$$\texttt{f}_\texttt{k}\ \texttt{(MkR a}_1\ \ldots\ \texttt{a}_\texttt{n}\texttt{) = a}_\texttt{k}$$

$$\texttt{set\_f}_k \texttt{ : t}_\texttt{k}\texttt{ -> R -> R}$$
$$\texttt{f}_\texttt{k}\ \texttt{a (MkR a}_1\ \ldots\ \texttt{a}_\texttt{n}\texttt{) = (MkR a}\ \ldots\ \texttt{a}_{\texttt{k}-1}\ \texttt{a a}_{\texttt{k}+1} \ldots \texttt{a}_\texttt{n}\texttt{)}$$

In our example type provider, record fields can be either strings, floating-point numbers, or arbitrary-precision integers. We represent these types using the following data type:

```
data Ty = STRING | INTEGER | DOUBLE
```

We use the type `List (String, Ty)` to represent the field names and types of a particular record type. A type provider to read a description from a text file looks like an ordinary Idris program:

```
getRec : String -> IO (Provider (List (String, Ty)))
getRec file =
    do (Right contents) <- readFile file
         | Left err => return (Error (show err))
       case parseStr rec contents of
         Left err => return (Error err)
         Right info => return (Provide info)
```

It attempts to read the contents of the given file, passing any errors on to the type provider error constructor. Then, it invokes a combinator parser on the contents of the specified file. Parse errors are returned as type provider errors, while successful results are returned in the `Provide` constructor to be spliced in.

The type provider can be invoked on a text file using the following syntax:

```
%provide (info : List (String, Ty))
     with getRec "ProviderTest.txt"
```

which results in the top-level name `info` being bound to the field specification in the text file, assuming that the file exists and is syntactically correct. The next step is to invoke an elaboration script to convert the field specification into a real Idris record type using the following syntax:

```
%runElab (genRecord `{{MyRecord}} info)
```

The elaboration script `genRecord` is responsible for generating the data type and its getters and setters. It is invoked with a name and a record specification as arguments.

```
genRecord : TTName -> List (String, Ty) -> Elab ()
genRecord n fields =
  do recN <- inThisNS n
     declareDatatype $ Declare recN [] `(Type)
     ctorN <- ctorName recN
     let tyDefn = DefineDatatype recN [
       Constructor ctorN
                   (map fieldArg fields)
                   (Var recN)
     ]
     defineDatatype tyDefn
     mkAccessors tyDefn
```

The first step in `genRecord` is to place the users's provided name into the current namespace. Then, it declares a zero-argument data type with the namespaced name. After that, it generates a name for the constructor using `ctorName`, which prepends `Mk` to the user-provided record type name. Next, it adds the constructor, using the helper function `fieldArg` to convert `(String, Ty)` pairs into the `FunArg` type described in Section 3.3. Finally, it adds the constructor to the data type and calls a helper operation to define the accessors according to the scheme described earlier in this section. Having done this, the type is now an ordinary Idris record type and can be used with Idris's record syntax. Additionally, it will be compiled just like any other Idris record type.

6. Related Work

Reflection in Agda (pre-2016) Agda's elaborator is different in a number of ways from Idris's: the core language is not strictly delineated from the high-level language and a global metavariable context is used instead of an Oleg-style development calculus (Norell 2007). Agda's type checker can be seen as an elaborator from terms that may contain metavariables to terms that do not. In previous versions of Agda, a number of quoting and unquoting splicing operators were available, along with a data type representing reified high-level Agda terms and definitions. Because the splicing operator acted on high-level Agda terms, it could perform elaboration to

do things like solving implicit arguments. Previous work by van der Walt and Swierstra (2012) and Kokke and Swierstra (2015) demonstrated how to use this to encode non-trivial proof automation in Agda itself, and both cite the fact that this proof automation is performed within Agda itself as major advantages. Unfortunately, both systems needed to reimplement much of the infrastructure that is part of the Agda implementation, such as unification, and each was forced to duplicate information about the types of values that Agda already has in its global context. These disadvantages of indirect approaches to reflection are described by Barzilay (2006).

Reflection in Agda (post-2016) Inspired by Idris's elaborator reflection, the Agda language has recently adopted a form of elaborator reflection in place of the previous API[3]. Since version 2.5.1, Agda exposes its elaboration monad, including the unification and metavariable machinery, to user elaboration scripts. Because the details of its elaboration mechanism differ substantially from those of Idris, so do the operations exposed in its reflected elaborator.

Direct vs. Indirect Reflection In his Ph.D. thesis on reflection in Nuprl, Barzilay (2006) contrasts two approaches to the design of a reflection system: *indirect reflection*, in which the aspects of the language that are to be reflected are re-implemented internally, and *direct reflection*, in which the underlying implementation is invoked directly. Barzilay points out a number of advantages of the direct approach to reflection: indirect representations of terms have an exponential increase in memory usage as levels of quotation increase, it becomes more difficult to ensure the correspondence between both implementations of the system, and having two implementations of each feature vastly increases the work that is necessary to develop and maintain new language features.

While Idris's reflection system uses an indirect representation of reified terms, the exponential blowup is not a major problem because typical applications require no more than one level of quotation. Additionally, because the quoted terms are in Idris's small, stable core language, concerns about ongoing development are less pressing than they would be if the high-level language were quoted. The large, complex parts of Idris, such as the type checker and the elaboration machinery, are reflected directly. For example, the elaboration strategy for Idris's `rewrite ... in ...` syntax, which rewrites the current goal type using an equality proof, is exposed directly as the `rewriteWith` primitive, rather than requiring that it be re-implemented in Idris code. Likewise, the `apply` tactic, which has proven to be highly useful in the elaborator for Idris that is written in Haskell, is exposed directly rather than being implemented in a library. Just as it is in the Haskell elaborator, `apply` has proven to be greatly valuable in the reflected elaborator. This tactic requires somewhat complex reimplementations of the typing rules for dependent functions in order to compute the correct dependent types for the holes that the operator will be applied to, and it makes little sense not to share the implementation.

Template Haskell Template Haskell (Sheard and Jones 2002) is a metaprogramming system for Haskell in which metaprograms are monadic actions that transform Haskell programs. It has been enabled a great deal of interesting work, including prototype implementations of generic programming extensions (Norell and Jansson 2004). Template Haskell's `Q` monad, however, does not supply effects related to the elaboration infrastructure of implementations such as GHC. Instead, it provides only the ability to produce fresh names and the ability to look up definitions from the global context.

Tactic-Based Interactive Proof Assistants There is a rich tradition of interactive proof assistants based on automation through tactic languages. Indeed, the ML family of programming languages began as a tactic language for the Edinburgh LCF system, and Nuprl also uses a dialect of ML for proof automation. Coq has a dedicated DSL for writing tactics, known as LTac (Delahaye 2000), with features such as backtracking and matching against terms and contexts. Idris's elaboration infrastructure intentionally resembles the control structures and features found in these systems. However, the reflected elaborator uses Idris as its own metalanguage, allowing code re-use across programs, specifications, and proofs automation or metaprogramming.

MTac MTac (Ziliani et al. 2013) is an alternative tactic language for Coq that, like `Elab`, provides a monadic interface to proof automation, using Coq's own term language. In MTac, a computation with type `M t` is a tactic that will either solve a goal of type `t` or fail. In other words, the type of a tactic provides information about the goals to which it can be applied. While they are superficially similar, MTac and elaborator reflection serve different purposes. MTac is a dedicated proof automation language, specifically tailored to that purpose, while elaborator reflection is intended to be a general-purpose means of language extension. Implementing MTac's primitives would be an interesting use of the reflected elaborator.

Internal Metaprogramming in Type Theory Altenkirch and McBride (2003) and Benke et al. (2003) pointed out that many applications of generic programming can be internalized directly into type theory, using a universe encoding. Later work by Chapman et al. (2010) eshowed that primitive facilities for defining data types can be replaced by a single self-describing type of descriptions, unifying generic programming with ordinary programming. Chapman et al. refer to this technique as *levitation*. Inspiring as this work is, many practical problems remain to be solved before we can build our languages on top of levitation. In particular, we do not yet have predictable optimization strategies to eliminate all extraneous data type descriptions at run time, and implementations of techniques like levitation are notoriously slow when compared to native data types in existing type checkers. While we wait for the missing technology, other techniques are relevant and interesting.

Other, special-purpose universes have been used to write non-trivial and interesting metaprograms, including some of the ones that we have implemented with elaborator reflection, internally to type theory. In particular, McBride (2015) showed how to generically describe general recursion in Agda, including a generic version of the Bove-Capretta predicates that are implemented using an encoding based on Dybjer and Setzer's 1999 encoding of inductive-recursive definitions. Devriese and Piessens (2013) provided a large collection of metaprogramming tools implemented directly within Agda. Because these metaprograms are written in the type theory, they are also verified, and they have all the other advantages of programs written in type theory. While this line of work is exciting, there are many important problems to be solved before we can rely on internalized reasoning for all of our metaprogramming needs. First, described data types still have a significant performance overhead, both during type checking and execution, relative to native data types and definitions. Second, the need to account for every detail internally can lead to epic amounts of code and work to automate even simple procedures — Devriese and Piessens's metaprogramming system required approximately 1200 lines of code to implement a generic printer, much of which is a mandatory correctness proof for the metaprogram. By contrast, programs in the reflected elaborator are similar in length and complexity to traditional tactic scripts and code generators. An implementation of type class instance derivation for `Show` using Idris's elaborator reflection that handles more data types than Devriese and Piessens's implementation and must take care of additional details like generating instance objects is less than 300 lines of code. Finally, a number of separate

[3] Documented on a post to the Agda mailing list: `https://lists.chalmers.se/pipermail/agda/2016/008414.html` and in the release notes for Agda version 2.5.1

universes for metaprogramming exist, with different strengths and weaknesses, and code written for one is not applicable to the others.

While one of these universes or tools could be given special syntactic and run-time support, privileging a single metaprogramming system for fast execution and tool support might prevent progress with others. Additionally, metaprograms written in `Elab` can use verified internal metaprograms' tactics by generating the appropriate code. We need not make an all-or-nothing choice between external and internal metaprogramming, and elaborator reflection could even be used to ease some of the overhead of using internal metaprogramming features, just as Idris's elaborator eases the burden of working in a very small language like TT.

7. Discussion

We have described an approach to metaprogramming called *elaborator reflection* that is applicable to languages that have a type checker that synthesizes terms in either a separate core language or a much more explicit version of the source language. In Idris, elaborator reflection has enabled features to be moved from the compiler to libraries, making the implementation simpler and providing users with more flexibility. It has also replaced a previous special-purpose tactic language. The features provided by the elaboration framework, such as incremental construction of terms and higher-order unification, have proven to be useful for a variety of metaprogramming tasks. Additionally, elaborator reflection provides a framework within which other metaprogramming systems can be implemented.

In elaborator reflection, metaprograms have direct access to the infrastructure of the elaborator itself, which provides precisely those tools that were already necessary to elaborate the language in question. The concept of elaborator reflection is not limited to implementations of dependent type theory. It is applicable to any language in which concepts outside of the language's own semantics are used to implement a type checker or elaborator. While Agda has already adopted elaborator reflection, it is instructive to consider how the concept might look in other languages.

The GHC implementation of Haskell performs elaboration by arranging for a constraint solver to produce explicit evidence in a small core language, System FC_2. Initially, a program is traversed, generating a large collection of constraints to be solved. When these constraints are solved, each solution contributes to the final program in the core language, which can be type checked independently (Vytiniotis et al. 2011). It could be interesting to design a reflection of this system, providing Haskell code with the ability to add new constraints with novel means of creating evidence terms. This could serve as an alternative to more "heavyweight" constructions such as the type checker plugin API.

7.1 Future Work

Elaboration Stages Like Template Haskell, Idris's elaborator reflection system suffers from subtle staging issues. It is convenient to be able to define an elaboration script and use it in the same source file. On the other hand, the close dependence of some elaboration scripts on their own elaboration order may lead to reluctance to change the elaborator in the future, due to the risk of breaking existing metaprograms. A potential solution to these issues can be found in Flatt's 2002 paper about the Racket (then MzScheme) module system and its treatment of macros. In this module system, the visibility of names is stratified into metalevels, with compile-time macro code being inaccessible to run-time code and *vice versa*. Additionally, due to the possibility of metaprograms that generate or manipulate metaprograms, there is an infinite hierarchy of stages. Within a module definition, components can be defined or imported at any stage. A similar addition to the Idris module system that classifies imports and definitions by their elaboration stage could make it possible to reliably reason about elaboration times and statically reject metaprograms that do not respect staging.

Generic Programming Unlike Haskell, Idris does not have a "native" type class deriving feature. `Elab` scripts can be used to implement non-trivial type class deriving, but the level of abstraction is quite low. Generic deriving mechanisms that use a uniform representation of data types, such as that described by Magalhães et al. (2010) and implemented in GHC, provide a more elegant approach that additionally allows generic programs to be written as ordinary programs, rather than through code generation. Similarly, dependent types allow non-trivial data types to be encoded generically, but the dust has not yet settled around which universe encoding of data types is best. Norell and Jansson (2004) describe an application of Template Haskell to implementing novel generic programming systems. Similarly, the Shapeless generic programming library for Scala[4] uses Scala's macro facility to implement its generics, an approach discussed by Burmako's (2013). We would like to follow the lead of Norell and Jansson and the Shapeless developers and use elaborator reflection to make an implementation of a high-level generic programming system convenient to use with native data types.

Alternative Proof Languages The reflected elaborator provides many useful primitives, but exposes a very low-level interface for proof construction. Users must keep track of individual holes, ensuring that each of them is filled and solved in turn. Matching against reflected terms is highly intensional, not even respecting Idris's definitional equality, and normalization must be specifically requested. Additionally, a single proof automation API is not equally suitable for all tasks. In future, we would like to implement higher-level proof APIs by interpreting them in `Elab`. We have already implemented a significant portion of Agda's former reflection API, and we would like to explore an implementation of MTac.

Self Hosting Thus far, the only non-trivial dependently typed language implemented in itself is F* (Swamy et al. 2016). F* was bootstrapped by writing the first version in a subset of the language that overlaps with F#, using the F# compiler to compile the initial version. Rather than implementing a direct compiler in the style of Idris, F* is first translated to either F# or OCaml, which allowed subsequent versions of F* to be compiled using the respective compilers. Elaborator reflection suggests an alternate route to self-hosting. Given facilities for syntactic abstraction and the ability to write terms directly in the core language, the elaborators for a high-level language like Idris could be written as macros, expanding to the core language. Then, the only primitives needed for bootstrapping would be an implementation of the core type theory, the macro system, and the basic elaboration primitives.

Elaborator reflection empowers Idris's users to take charge of their language. In a self-hosted Idris, implemented by means of a tower of reflected elaborators with a stage-respecting module system, users will have freedom comparable to that of a Lisp user. At the same time, the safety and stability of the TT type checker provides a solid grounding for new features, ensuring that they have a sensible explanation. A reflected elaborator on top of type theory provides both freedom and security.

References

T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Proceedings of IFIP TC2/WG2.1 Working Conference on Generic Programming*, 2003.

E. Barzilay. *Implementing Reflection in Nuprl*. PhD thesis, Cornell University, 2006.

[4] Available from `https://github.com/milessabin/shapeless`

M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4): 265–289, Dec. 2003.

A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, Aug. 2005.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

E. Brady and K. Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In P. Hudak and S. Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 297–308. ACM, 2010. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863587. URL `http://doi.acm.org/10.1145/1863543.1863587`.

E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2003. ISBN 3-540-22164-6. doi: 10.1007/978-3-540-24849-1_8. URL `http://dx.doi.org/10.1007/978-3-540-24849-1_8`.

E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*. ACM, 2013.

J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 3–14, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863547. URL `http://doi.acm.org/10.1145/1863543.1863547`.

A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. URL {`http://adam.chlipala.net/cpdt/`}. Available online: `http://adam.chlipala.net/cpdt/`.

D. R. Christiansen. Dependent type providers. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, WGP '13, pages 25–34, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2389-5. doi: 10.1145/2502488.2502495. URL `http://doi.acm.org/10.1145/2502488.2502495`.

D. R. Christiansen. Type-directed elaboration of quasiquotations: A high-level syntax for low-level reflection. In *Proceedings of the 26th International Symposium on Implementation and Application of Functional Languages*, IFL '14, October 2014.

D. Delahaye. A tactic language for the system coq. In *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, LPAR'00, pages 85–95, Berlin, Heidelberg, 2000. Springer-Verlag.

D. Devriese and F. Piessens. Typed syntactic meta-programming. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 73–86, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500575. URL `http://doi.acm.org/10.1145/2500365.2500575`.

P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *The Journal of Symbolic Logic*, 65:525–549, 6 2000. ISSN 1943-5886. doi: 10.2307/2586554.

P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In J. Girard, editor, *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer, 1999. ISBN 3-540-65763-0. doi: 10.1007/3-540-48959-2_11. URL `http://dx.doi.org/10.1007/3-540-48959-2_11`.

M. Flatt. Composable and compilable macros:: You want it when? In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 72–83, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8. doi: 10.1145/581478.581486. URL `http://doi.acm.org/10.1145/581478.581486`.

P. Kokke and W. Swierstra. Auto in agda. In R. Hinze and J. Voigtlnder, editors, *Mathematics of Program Construction*, volume 9129 of *Lecture Notes in Computer Science*, pages 276–301. Springer International Publishing, 2015. ISBN 978-3-319-19796-8. doi: 10.1007/978-3-319-19797-5_14. URL `http://dx.doi.org/10.1007/978-3-319-19797-5_14`.

J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 37–48, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. doi: 10.1145/1863523.1863529. URL `http://doi.acm.org/10.1145/1863523.1863529`.

C. McBride. *Dependently typed programs and their proofs*. PhD thesis, University of Edinburgh, 1999.

C. McBride. Turing-completeness totally free. In R. Hinze and J. Voigtländer, editors, *Proceedings of Mathematics of Program Construction: 12th International Conference, MPC 2015, Königswinter, Germany, June 29–July 1, 2015*, pages 257–275. Springer International Publishing, 2015. ISBN 978-3-319-19797-5.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

U. Norell and P. Jansson. Prototyping generic programming in Template Haskell. In D. Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 314–333. Springer-Verlag, 2004.

T. Sheard and S. P. Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 1–16. ACM, 2002. ISBN 1-58113-605-6.

N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 256–270, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837655. URL `http://doi.acm.org/10.1145/2837614.2837655`.

D. Syme, K. Battocchi, K. Takeda, D. Malayeri, and T. Petricek. Themes in information-rich functional programming for internet-scale data sources. In *Proceedings Workshop on Data Driven Functional Programming*, DDFP 2013, 2013.

P. van der Walt and W. Swierstra. Engineering proof by reflection in agda. In R. Hinze, editor, *24th International Symposium on Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-41582-1_10.

D. Vytiniotis, S. L. P. Jones, T. Schrijvers, and M. Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21 (4-5):333–412, 2011. doi: 10.1017/S0956796811000098. URL `http://dx.doi.org/10.1017/S0956796811000098`.

B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: A monad for typed tactic programming in Coq. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 87–100. ACM, 2013.