— *Draft submitted to TFP 2014 post-proceedings* —

Reflect on Your Mistakes!
Lightweight Domain-Specific Error Messages

David Raymond Christiansen

IT University of Copenhagen
drc@itu.dk

Abstract. Embedded domain-specific languages allow the rapid prototyping and development of new languages. Host languages with rich type systems can encode a wide variety of embedded language type systems, leading to the same static safety guarantees that a stand-alone implementation would enjoy. However, the error messages generated by these embedded type systems are notoriously difficult to understand, which can limit the practical utility of safe embedded languages. This paper presents a language feature, called *error reflection*, that allows EDSL and library authors to customize the display of compiler errors based on reflected representations of these errors. Additionally, error reflection is applicable to other situations in which immediate surface syntax does not correspond to underlying representations.

1 Introduction

Much of the discourse about domain-specific languages (DSLs) has traditionally focused on allowing domain experts, rather than professional software developers, to describe non-trivial software in a notation that is close to their mental models. Embedded domain-specific languages (EDSLs) were described by Hudak in 1996 [11]. In an EDSL, an already-existing general-purpose language is used to express the constructs of the domain-specific language, which may not require as large of an investment of time as a stand-alone implementation due to re-use of existing development tools. In recent years, another category of embedded domain-specific languages has come to the fore: EDSLs intended for professional software developers that mask some of the complexity of a particular area of programming. Examples of this category include Chafi et al.'s work with heterogeneous parallelism [7], the Repa array language [12], and the Feldspar DSP language [1].

Embedding DSLs in a host language with an expressive type system allows domain-specific type systems to be encoded in the host type system. This has a number of benefits: the DSL type system inherits properties from the host language such as type soundness and decidability of type checking, the DSL developers do not need to implement complicated features such as type inference, and existing development tools can support the embedded language's type system without further extension or customization. The Achilles' heel of this

approach is the convoluted error messages that can result from non-trivial encodings: they may bear little resemblance to the embedded surface syntax and they may be long and stereotypical. The utility of a type error that no user can understand is questionable at best.

Idris [3] is a dependently typed functional programming language that is expressly built to be a good host for embedded languages. Examples of non-trivial embedded languages include Brady's algebraic effects language [4] and his earlier language for resource-safe programming [5]. Full dependent types, where the type system includes the full computational power of the total fragment of the term language, have the potential to be a host for very expressive embedded type systems. Additionally, embedded languages have the potential to hide the complexity of full dependent types, allowing programmers to write code as if they were working in a simpler language. If this goal is to be practical, then the problem of error messages must be solved.

An embedded query language for relational databases might require that projections from tuples select columns that actually exist in the schema. The mechanism described in this paper can transform this generic error message:

```
Can't solve goal
        HasCol [("name", STRING), ("age", INT)] "naime" STRING
```

into this domain-specific error message:

```
The schema [("name", STRING), ("age", INT)] does not contain the
column "naime" with type STRING
```

Contributions

The primary contribution of this paper is a novel extension to the Idris language, called *error reflection*, that expands the scope of the reflection mechanism used for proof automation to encompass error reporting. Error reflection enables developers to rewrite the compile-time error messages that result from complex APIs, particularly embedded domain-specific languages, so that the error messages can be consistent with the ideas and metaphors of the API or DSL. The paper presents several examples of error messages that can be improved using error reflection. Though the technique is implemented in Idris, there is reason to believe that it would be applicable to languages without dependent types. Error reflection is implemented in versions of Idris numbered 0.9.13 and higher.

2 Background

2.1 Reflection in Idris

Idris's reflection mechanism is closely related to that of Agda [22]. The reflection mechanism defines a datatype that represents expressions, along with a means for quoting, or *reflecting*, expressions to this datatype and, conversely, a means

of *reifying* these representations to real expressions. This allows the language itself to be used to automate the construction of proofs.

As described by Brady [3], the Idris compiler contains two programming languages: a high-level user-friendly functional language and a low-level fully-explicit core language that is easy for the machine to type check. A process called *elaboration* explains the features of the high-level language, such as type classes, implicit arguments, and `where`-blocks, in terms of the simple core language.

Idris's reflection datatype faithfully represents the expressions of this core language. The original use for reflection was the implementation of custom proof tactics, represented as functions from reflected representations of a proof context and goal to a tactic script that solves this goal. Reflection has not yet seen major use as a means of compile-time code generation in the style of Template Haskell [18].

2.2 Idris Quasiquotation

Reflected terms faithfully represent Idris's highly explicit core theory, and are therefore somewhat awkward to destructure and construct. An experimental extension to Idris supports the use of *quasiquotations* for manipulating reflected terms. Quasiquotations, which syntactically consist of parenthesized Idris terms preceded by a back-tick, represent the reflected version of the term in the quotation. Antiquotations, which are subterms preceded by a tilde,[1] cause the antiquoted reflected term to be spliced in to the surrounding quotation. When pattern matching, antiquotations instead contain other patterns, which can be either pattern variables or patterns that match reflected terms.

A proof script tactic similar to Coq's `reflexivity`, which attempts to solve the current goal with the constructor of the equality type, can be implemented using quasiquotation patterns as follows:

```
reflexivity : List (TTName, Binder TT) -> TT -> Tactic
reflexivity _ `(~x = ~y) = Refine "refl"
reflexivity _ _          = Search 0
```

The tactic checks whether the goal is an equality type, and if so, applies the constructor. In other cases, it performs a zero-depth proof search, i.e., nothing.

This notion of quasiquotation is closer to those found in the Lisp family [2] and Scala [17] rather than to those found in Camlp4 [16] and Haskell [13], because Idris quasiquotations are used primarily for manipulating Idris code rather than for embedding custom notations. Idris quasiquotations will be described in detail in a forthcoming publication. Quasiquotations are not yet available in a released version of Idris, but they will be in the release following version 0.9.13.1.

2.3 Techniques for Embedding Languages

Shallow embeddings of DSLs directly use the features of the host language where they coincide with the embedded language. For example, addition in the embed-

[1] The specific syntax is inspired by the Clojure dialect of Lisp [10].

ded language might directly correspond to addition in the host language, and the variable binding mechanisms of the host language can implement binders in the embedded language (a technique known as higher-order abstract syntax, or HOAS [15]). Shallow embeddings can often provide very convenient interfaces, but they have important drawbacks: there is no direct representation of the language's syntax for later processing, and typical representations of HOAS lead to datatypes that are not strictly positive, rendering them unfit for use in dependently typed languages.

A deep embedding uses an ordinary datatype to represent the abstract syntax of the embedded language. Indexed families, as found in post-GADT Haskell and OCaml as well as in dependently typed languages, allow this approach to encode expressive typing rules. Additionally, Idris has feature called *DSL notation* [5] that allows the direct reification of Idris's binding syntax to de Bruijn-indexed datatype families with any inferrable collection of indices, which removes much of the syntactic advantage of shallow embeddings. However, even though indexed families and DSL notation provide an expressive framework for describing embedded DSLs in a safe and convenient manner, not every feature of an embedded language is necessarily convenient to represent in this notation.

Many embedded languages do not fit the general framework of a functional programming language's type system. Quite often, an embedded language will have "side conditions" — invariants on code that should be statically checked. Examples might include ensuring that a variable name is found in some context or that all elements of one list are contained in another. There are various means of encoding these constraints, such as Haskell's type classes and Scala's implicit arguments. In Idris, a commonly-used technique is to define a type whose inhabitants witness the side condition in question, and then arrange for the compiler to automatically construct these witnesses when possible. In the interest of simplicity, this paper does not use the more advanced methods of language embedding. However, improving error messages that result from checking side conditions is an important success criterion for the language feature presented in this paper.

3 Error Reflection

3.1 Motivating Example

As a very simple motivating example, consider a tiny fragment of a database interface library that contains schemas, tuples, and relations, along with Cartesian products of relations and projection of individual elements from tuples. Following Oury and Swierstra [14], we define a simple universe of datatypes that will be supported in the database. For the sake of simplicity, the embedded language will support only integers and strings:

```
data Ty = INT | STRING

interpTy : Ty -> Type
interpTy INT    = Int
interpTy STRING = String
```

Schemas are simply lists of pairs of column names and codes from the universe:

```
Schema : Type
Schema = List (String, Ty)
```

Tuples are represented by the family `Row`, which is indexed by schemas:

```
data Row : Schema -> Type where
  Nil : Row []
  (::) : interpTy t -> Row s -> Row ((c,t) :: s)
```

In Idris, naming the constructors `Nil` and `(::)` allows list literal syntax to be used to construct a `Row`. The following listing puts these pieces together, showing a concrete schema and row:

```
r : Row [("name", STRING), ("age", INT)]
r = ["Jane", 43]
```

Projections from a tuple are slightly more complicated: the type of the result depends on the schema, and the system should disallow projections of columns that do not exist in the schema. Preferably, this check will occur statically. A convenient way to achieve this is to define a type of witnesses that a particular attribute is present in a schema, and then arrange for Idris to construct these witnesses on demand. The type `HasCol s c t` represents that schema `s` contains a column named `c` with type `t`, using the standard technique.

```
data HasCol : Schema -> String -> Ty -> Type where
  Here : HasCol ((c, t) :: s) c t
  There : HasCol s c t -> HasCol ((c',t')::s) c t
```

Projection can now be defined by recursion over the structure of these witnesses:

```
project : (c : String) -> (t : Ty) -> (r : Row s) ->
          (ok : HasCol s c t) ->
          interpTy t
project c t []        ok        = FalseElim (emptyNoCols ok)
project c t (x :: xs) Here      = x
project c t (x :: xs) (There p) = project c t xs p
```

In practice, however, we cannot expect users of our embedded query language to construct a `HasCol` every time they want to project an element from a tuple. Thus, we redefine `ok` to be an implicit argument that should be inferred by the compiler. The `auto` keyword causes Idris to construct the `HasCol` witness using its built-in proof search.

```
project : (c : String) -> (t : Ty) -> (r : Row s) ->
          {auto ok : HasCol s c t} ->
          interpTy t
project c t []        {ok = ok}      = FalseElim (emptyNoCols ok)
project c t (x :: xs) {ok = Here}    = x
project c t (x :: xs) {ok = There p} = project c t xs {ok=p}
```

A *relation* is a collection of tuples with the same schema. Here, a relation containing `n` tuples with schema `s` is represented by a `Vect n (Row s)`. Like SQL and unlike the relational algebra, this encoding allows for duplicate tuples. The Cartesian product, written here with the `(*)` operator, is the concatenation of each row from one relation with each row from another. Just like projection, the Cartesian product has a side condition: it is only defined for tuples whose collection of attribute names are disjoint. As before, we represent this side condition using an automatically-solved implicit proof of disjointness.

```
(*) : Vect n (Row s1) -> Vect m (Row s2) ->
      {auto prf : Disjoint s1 s2} ->
      Vect (n * m) (Row (s1 ++ s2))
(*) []      ys             = []
(*) (r::rs) ys {prf = prf} = map (r++) ys ++ ((*) rs ys {prf=prf})
```

Ideally, users of an embedded database language would be able to work entirely within the abstractions of the database language, rather than needing to worry about the details of proof automation and implicit arguments. However, when they make mistakes, the error messages are expressed in terms of the underlying implementation of the static semantics. The purpose of this work is to improve on this situation.

3.2 Error Messages

The relation `humans` describes people and their ages, while `housing` lists the size of two apartments:

```
humans : Vect 2 (Row [("name", STRING), ("age", INT)])
humans = [ ["Alice", 37], ["Bob", 23]]

housing : Vect  2 (Row [("floorspace", INT)])
housing = [[48], [72]]
```

The first column of the first row in `humans` can be extracted using `project`, but if a user misspells a column name, then the resulting error message can be difficult to decode:

```
> project "name" STRING (head humans)
"Alice" : String

> project "naime" STRING (head humans)
Can't solve goal
        HasCol [("name", STRING), ("age", INT)] "naime" STRING
```

Likewise, the Cartesian product of `humans` and `housing` contains the expected four tuples. However, trying to take the product of `humans` and itself results in an error that reflects details of the DSL implementation rather than domain concepts:

```
> humans * housing
[["Alice", 37, 48],
 ["Alice", 37, 72],
 ["Bob", 23, 48],
 ["Bob", 23, 72]] : Vect 4
                          (Row [("name", STRING),
                                ("age", INT),
                                ("floorspace", INT)])

> humans * humans
Can't solve goal
        Disjoint [("name", STRING), ("age", INT)]
                 [("name", STRING), ("age", INT)]
```

Careful naming of the required proof objects can sometimes lead to these errors being somewhat understandable, as above. However, good error messages should do more than simply provide a vague hint about why a problem arose. They should explain the problem, and do so in an accessible and straightforward manner.

3.3 Reflecting Errors

In recent versions of Idris, the compiler is capable of reflecting its error messages. The standard library contains a datatype corresponding to the compiler's own internal representation of errors, and Idris functions can insert themselves into the error reporting mechanism to rewrite errors before they are shown to users. In some sense, these error handlers resemble exception handlers, except they can only raise a new exception, rather than recovering from the error and continuing the program.

An error handler is a partial function from a representation `Err` of error messages to a rewritten error report. Accordingly, we might assign them the type `Err -> Maybe String`. However, error messages have more structure than a String can express. Often, they will include Idris terms, or have a hierarchical structure. Error reports that result from reflection should be able to use the facilities of the compiler that already exist for rendering this structure. Thus, Idris defines a type ErrorReportPart that represents the various sorts of content that can appear in an error report.

```
data ErrorReportPart = TextPart String
                     | NamePart TTName
                     | TermPart TT
                     | SubReport (List ErrorReportPart)
```

The constructor `TextPart` represents a string containing error explanations, `NamePart` contains a reflected Idris name to be highlighted, `TermPart` contains a reflected Idris term to be pretty-printed, and `SubReport` contains a report to be displayed as sub-details of a report.

The representation of errors `Err` is simply a subset of the constructors of the compiler's internal datatype that represents errors, including the most important errors, such as conversion errors, unification errors, and proof search failures. It is a subset because error reflection should not rewrite errors that have already been rewritten, and because some types of errors exist primarily to keep track of things like source location, which rewritten errors should not lie about.

Error handlers map reflected errors to lists of these error report parts. Because not every handler will handle every error, error handlers should have the type `Err -> Maybe (List ErrorReportPart)`. To avoid the accidental application of an error handler, the compiler pragma `%error_handler` is used to mark functions with this type as error handlers.

The function `dbErr` in Figure 1 maps proof search errors in the above code to domain-specific error messages. It relies on an auxiliary function `getHasColFields`, which simply extracts the three parameters of the reflected representation of a `HasCol` proof.

As we saw in Section 3.1, the following definition quite obviously fails to satisfy the condition that the arguments of the Cartesian product of relations should be disjoint:

```
test3 : Vect 4 (Row [("name", STRING), ("age", INT),
                     ("name", STRING), ("age", INT)])
test3 = humans * humans
```

Now, however, the resulting error message refers specifically to the notion of disjointness:

```
When elaborating right hand side of test3:
The schemas [("name", STRING), ("age", INT)] and
[("name", STRING), ("age", INT)] are not disjoint.
```

Additionally, misspelling a column name when performing a projection provides a clear error. The definition:

```
floorspace : Int
floorspace = project "flodorspace" INT (index 2 (humans * housing))
```

yields the error

```
When elaborating right hand side of floorspace:
The schema [("name", STRING), ("age", INT)] ++ [("floorspace", INT)]
does not contain the column "flodorspace" with type INT
```

which quite straightforwardly explains the problem.

4 Case Studies

While error reflection was a useful technique for improving the usability of our fragment of a domain-specific language, we should hope that the applicability is somewhat broader. This section exhibits two specific applications for error reflection in already-existing Idris code.

```
%error_handler
total
dbErr : Err -> Maybe (List ErrorReportPart)
dbErr (CantSolveGoal `(Disjoint ~s1 ~s2) _) =
  Just [ TextPart "The schemas", TermPart s1
       , TextPart "and", TermPart s2
       , TextPart "are not disjoint." ]
dbErr (CantSolveGoal `(HasCol ~s ~c ~(P Bound _ _)) _) =
  Just [ TextPart "The schema", TermPart s
       , TextPart "does not contain the column"
       , TermPart c
       ]
dbErr (CantSolveGoal `(HasCol ~s ~c ~t) _) =
  Just [ TextPart "The schema", TermPart s
       , TextPart "does not contain the column"
       , TermPart c, TextPart "with type"
       , TermPart t
       ]
dbErr _ = Nothing
```

Fig. 1. An error handler for the embedded database language. Quasiquotations are used to destructure the terms inside of reflected errors.

4.1 Finite Set Literals

Error reflection is useful for more than just embedded DSLs. The Idris standard library contains a number of potential sources of confusing error messages. Error reflection can be used to bring these closer to what is expected.

A typical example of dependent types are the finite sets. Given a natural number `n`, the type `Fin n` has exactly `n` elements. In other words, `Fin 0` is uninhabited, `Fin 1` has precisely one element, `Fin 2` has precisely two elements, and so forth. The `Fin` family is often used for bounds-checked indexing into data structures. Similarly to Haskell, when Idris encounters an integer literal (say, `42`), it is desugared to `fromInteger 42`. Unlike Haskell, type-driven ad hoc overloading is used to disambiguate `fromInteger`. The Idris prelude contains the following definition:

```
fromInteger : (x : Integer) ->
              {default ItIsJust
               prf : (IsJust (integerToFin x n))} -> Fin n
fromInteger {n} x {prf} with (integerToFin x n)
  fromInteger {n} x {prf = ItIsJust} | Just y = y
```

Here, `IsJust : Maybe a -> Type` is a family of proofs that their arguments are built with the `Just` constructor and the `default ItIsJust` modification to the implicit argument causes Idris to use the constructor of these proofs to solve it. The function `integerToFin` simply returns the corresponding `Fin` if possible, or

`Nothing` otherwise. This combination, then, statically ensures that `Fin` literals are within their bounds.

Unfortunately, the error message that results from this arrangement is somewhat opaque. The simple definition:

```
f : Fin 2
f = 3
```

results in a quite involved error, in which otherwise-hidden details of the implementation take center stage:

```
When elaborating right hand side of f:
     Can't unify
             IsJust (Just x)
     with
             IsJust (integerToFin 3 2)

     Specifically:
             Can't unify
                     Just x
             with
                     Nothing
```

It is straightforward to define an error handler that will rewrite this error to something understandable. The error handler is demonstrated in Figure 2.

```
%error_handler
finTooBig : Err -> Maybe (List ErrorReportPart)
finTooBig (CantUnify x tm `(IsJust (integerToFin ~n ~m)) err xs y)
  = Just [ TextPart "When using" , TermPart n
         , TextPart "as a literal for a"
         , TermPart `(Fin ~m)
         , SubReport [ TextPart "Could not show that"
                     , TermPart n
                     , TextPart "is less than"
                     , TermPart m
                     ]
         ]
finTooBig _ = Nothing
```

Fig. 2. An error handler for finite set literals

In the presence of this error handler, the above definition of `f` results in a much more explanatory error message:

```
When elaborating right hand side of f:
     When using 3 as a literal for a Fin 2
             Could not show that 3 is less than 2
```

4.2 Algebraic Effects

Idris includes a library for handling side effects compositionally, without using monad transformers. An earlier version of this library is described in Brady's paper from ICFP 2013 [4]. Briefly, `{ [E, F r, G] } Eff m a` is the type of an effectful computation that uses effects `E`, `F`, and `G` in monad `m`, yielding a value in `a`. Additionally, the effect `F` has some resource that has type `r`.

When one operation in `Eff` calls another, the library searches for a proof that the called operation's effects are a subset of the calling operation's effects. This ensures that all effects that might be performed are visible in the caller's type, but it does not require that the called operation have the exact same effect collection.

The following effectful program reads a name from standard input, and greets the user.

```
hello : { [STDIO] } Eff m ()
hello = do n <- getStr
           putStrLn ("Hello, " ++ n)
```

If this program is rewritten to read the name from a file, the type of `hello` must be updated. If it is not, as in the following program, then Idris will report a proof search error.

```
getName : { [FILE_IO ()] } Eff m (Maybe String)
getName = do ok <- open "test" Read
             case ok of
               False => return Nothing
               True => do name <- readLine
                          close
                          return (Just name)

hello : { [STDIO] } Eff m ()
hello = do n <- getName
           putStrLn ("Hello, " ++ n)
```

The error is:

```
When elaborating right hand side of hello:
     Can't solve goal
             SubList [(FILE_IO ())] [STDIO]
```

While a great deal of thought has gone into making this error as readable as possible, the effect system is just a library. The compiler can't possibly provide more useful suggestions.

An error handler can rewrite this to something more informative:

```
%error_handler
effErr : Err -> Maybe (List ErrorReportPart)
effErr (CantSolveGoal `(Effects.SubList ~required ~found) ctxt)
  = Just [ TextPart "Attempted to use an operation with effects"
         , TermPart required
         , TextPart "in a context where only"
         , TermPart found
         , TextPart "are available."]
effErr _ = Nothing
```

With this error handler, the message becomes:

```
When elaborating right hand side of hello:
Attempted to use an operation with effects [FILE_IO ()] in a context
where only [STDIO] are available.
```

This message provides the user with a much better hint as to the significance of the relationship between these lists.

5 Argument Error Handlers

The error reflection mechanism described thus far suffers from a major shortcoming: the risk of "false positives". For example, it is perfectly reasonable to expect that a library other than the effects library might use the `SubList` family and its associated proof search procedure. However, the error handler described in Section 4.2 will also be used to rewrite error messages resulting from this new library, giving blatantly misleading results. The risk of false positives arises when a single type is used for multiple purposes.

The problem could be solved by copying and pasting the definitions of the types in question to a separate namespace, and being careful about matching namespaces in error handlers. However, copying and pasting is not typically regarded as a good code re-use practice. Even worse, users may receive a very confusing message if a library developer is not aware that a particular type is used in more than one location. Developers of error handlers need not be the original authors of a library.

The chance of false positives can be reduced by narrowing the scope of error handlers. Thus, Idris supports attaching them to specific formal parameters of specific functions. A comma-separated list of error handler names `hs` is attached to parameter `x` of the function, constructor, or type constructor `f` using the pragma:

```
%error_handlers f x hs
```

When an error results from the elaboration of a term that occurs as an argument to f in the position indicated by x, the error handlers hs will be preferred over global error handlers.

6 Implementation considerations

While error reflection is implemented in Idris, there are no fundamental considerations that prevent it from being implemented in languages without dependent types. Nevertheless, a practical implementation requires a certain amount of compiler infrastructure that may not be available in every programming language.

Compile Time Evaluation Executing error handlers requires that the compiler be able to evaluate expressions while type checking. Thus, an interpreter for the language being type-checked should be available, and the type checker should have some facility for using it. This is available by definition in a dependently typed language, but many other languages will also be able to do this.

Ensuring Termination Running arbitrary code at compile time has the potential to cause the compiler to not terminate. Because dependently typed languages do this as a matter of course, they have evolved sophisticated techniques for ensuring that only terminating terms are evaluated at compile time. Some languages, such as Coq [21] and Agda [20], reject all terms that do not pass the termination checker. Others, such as Casinghino, Sjöberg, and Weirich's Zombie [6], have separate overlapping languages: a terminating language of logical formulae, which can run in the type checker, and a potentially non-terminating language for programs that will not occur in types. Idris checks all terms for termination, but potentially non-terminating terms are simply not reduced by the type checker. To preserve the termination of the type checking process, error handlers that do not pass the termination checker are rejected. If other languages adopt error reflection, they should also implement a termination checker, or handle non-termination through some other mechanism, such as a timeout or through QuickCheck-style [8] property-based testing. Additionally, because a termination checker does not guarantee speedy execution, timeouts and property-based testing may be useful even in the presence of termination checking.

Reflection Facilities In order to support error reflection, the reflection capabilities of a host language should, at a minimum, support reflection of both types and terms. In dependent types, this is trivial, as there is no syntactic distinction between types and other terms. In other systems, support for reflecting both syntactic categories can vary. Some systems, such as Template Haskell, support both straightforwardly. Some other languages, such as F# [19], have one system for quoting terms and another for representing types (namely, .NET reflection). Scala's quasiquotations [17] support both expressions and types, and could be a promising facility for implementing error reflection.

Error Origin Tracking In a dependently-typed language in which error handlers can be attached to specific function arguments, it is not sufficient to install error handlers as exception handlers in a traversal of the abstract syntax tree. This is because Idris's unifier accumulates a collection of unsolved unification problems,

which may become solvable by a mix of later unifications and reductions. At the end of elaboration, the compiler must check that no open unification problems remain. Thus, unification errors in particular may first be signaled far from their source. In Idris, this is addressed by annotating every error with a stack of surrounding applications, and then using this information to decide which error handlers are eligible to rewrite the error. As a side benefit, these location annotations can also provide ordinary error messages that are more informative.

7 Related Work

While the difficulties in interpreting error messages in embedded DSLs is well-known, there are comparatively few systems that attempt to address it. Here, we do not describe work on improving type errors in general, as the focus is on embedded languages.

Heeren, Hage, and Swierstra [9] present a system for constraint-based type inference in the context of the Helium language that aims at solving the same problems as Idris error reflection. Their system is defined at a higher level of abstraction, supporting the definition of custom typing rules that are then mechanically checked for consistency with the host language's type system. Additionally, their system supports defining "sibling functions" that are suggested as alternatives in the case of type errors. Finally, their system allows the order of type inference constraints to be controlled by library authors, making it easier to locate error messages at the real source of errors, rather than elsewhere in a library. Some of these techniques would be applicable in a language like Idris. In particular, sibling functions seem to be quite promising as a potential feature. However, there is no obvious way to apply these techniques to proof search failures, and checking that custom typing rules are a consequence of Idris's typing rules could require arbitrarily complicated computation due to dependent types. Additionally, the lack of global type inference in dependently-typed languages drastically reduces the utility of controlling the order in which constraints are checked.

8 Conclusion and Future Work

As demonstrated, the error reflection facility enables conversion of uninformative error messages to informative, domain-specific error messages. Quasiquote patterns enable a convenient syntax for destructuring terms that occur in reflected errors and reconstructing informative messages that contain terms.

However, there are still practical considerations to be worked out. Perhaps the most serious is the strong coupling between error handlers and the specific terms that occur in error messages. Both compiler updates and relatively small changes in an embedded language can cause fairly large changes in error messages. To make rewriting reflected errors more robust and flexible, it may be convenient to be able to use tools other than pattern matching to define error handlers. For

example, it might be possible to develop a sort of query language for reflected terms that allows convenient and expressive extraction of sub-terms.

It can also be difficult to mentally map the displayed error message to the constructor that represents it in the error type. This could be solved by integrating error reflection more closely into Idris's IDE support. For instance, an interactive command to reflect an error that is displayed on screen might make it easier to determine the structure to be rewritten.

Acknowledgments This work was performed as part of the Actulus project, funded by the Danish Advanced Technology Fund (*Højteknologifonden*) grant 017-2010-3. I would like to thank my advisor, Peter Sestoft, for his comments and feedback on this paper during its development, and Emil Axelsson and the anonymous reviewers for their useful suggestions. Additionally, Edwin Brady was very helpful with the Idris implementation, and John Hughes provided interesting thoughts about testing as a complement to or replacement for totality checking.

References

[1] Axelsson, E., Claessen, K., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajdax, A.: Feldspar: A domain specific language for digital signal processing algorithms. In: Formal Methods and Models for Codesign. pp. 169–178. MEMOCODE, IEEE (Jul 2010)

[2] Bawden, A.: Quasiquotation in Lisp. In: Partial Evaluation and Semantic-Based Program Manipulation. pp. 4–12. PEPM '99 (1999)

[3] Brady, E.: Idris, a general purpose dependently typed programming language: Design and implementation. Journal of Functional Programming 23, 552–593 (9 2013)

[4] Brady, E.: Programming and reasoning with algebraic effects and dependent types. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. pp. 133–144. ICFP '13, ACM (2013)

[5] Brady, E., Hammond, K.: Resource-safe systems programming with embedded domain specific languages. In: Russo, C., Zhou, N.F. (eds.) Practical Aspects of Declarative Languages, Lecture Notes in Computer Science, vol. 7149, pp. 242–257. Springer Berlin Heidelberg (2012)

[6] Casinghino, C., Sjöberg, V., Weirich, S.: Combining proofs and programs in a dependently typed langauge. In: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '14, San Diego, CA, USA (2014)

[7] Chafi, H., DeVito, Z., Moors, A., Rompf, T., Sujeeth, A.K., Hanrahan, P., Odersky, M., Olukotun, K.: Language virtualization for heterogeneous parallel computing. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. pp. 835–847. OOPSLA '10, ACM (2010)

[8] Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. pp. 268–279. ICFP '00, ACM (2000)
[9] Heeren, B., Hage, J., Swierstra, S.D.: Scripting the type inference process. In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming. pp. 3–13. ICFP '03, ACM (2003)
[10] Hickey, R.: Clojure (2008–2014), http://www.clojure.org/
[11] Hudak, P.: Building domain-specific embedded languages. ACM Computing Survey 28(4es) (Dec 1996)
[12] Keller, G., Chakravarty, M.M., Leshchinskiy, R., Peyton Jones, S., Lippmeier, B.: Regular, shape-polymorphic, parallel arrays in haskell. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. pp. 261–272. ICFP '10, ACM (2010)
[13] Mainland, G.: Why it's nice to be quoted: Quasiquoting for haskell. In: Proceedings of the ACM SIGPLAN Workshop on Haskell. pp. 73–82. Haskell '07, ACM (2007)
[14] Oury, N., Swierstra, W.: The power of Pi. In: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming. p. 39. ACM (Sep 2008)
[15] Pfenning, F., Elliot, C.: Higher-order abstract syntax. SIGPLAN Notices 23, 199–208 (1988)
[16] de Rauglaudre, D.: Camlp4 reference manual (2003), http://pauillac.inria.fr/camlp4/manual/
[17] Shabalin, D., Burmako, E., Odersky, M.: Quasiquotes for scala. Tech. Rep. 185242, École polytechnique fédérale de Lausanne (2013)
[18] Sheard, T., Jones, S.P.: Template meta-programming for haskell. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell. pp. 1–16. Haskell '02, ACM (2002)
[19] Syme, D.: Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In: Proceedings of the 2006 workshop on ML. pp. 43–54. ACM (2006)
[20] The Agda Team: The Agda Wiki (2014), http://wiki.portal.chalmers.se/agda/
[21] The Coq development team: The Coq proof assistant reference manual. LogiCal Project (2014), http://coq.inria.fr, version 8.4pl4
[22] van der Walt, P., Swierstra, W.: Engineering proof by reflection in Agda. In: Hinze, R. (ed.) Implementation and Application of Functional Languages. pp. 157–173. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2013)