# First-class Type-safe Reflection in Idris

Edwin Brady

School of Computer Science, University of St Andrews, St Andrews, Scotland.
ecb10@st-andrews.ac.uk

## Abstract

Idris is a general purpose purely functional programming language with dependent types, aiming to bring type-based program verification techniques to functional programmers. One common difficulty with programming with dependent types is that proof obligations arise naturally once programs become even moderately sized. For example, implementing an adder for binary numbers indexed over their natural number equivalents will naturally lead to proof obligations for equalities of expressions over natural numbers. Similarly, indexing a binary tree over its flattening as a list will naturally lead to proof obligations for associativity of list concatenation.

As far as possible, we would like to solve such proof obligations automatically. In this talk (which describes work in progress), I will show one way to automate such proofs by *reflection*. I will show how representing Idris expressions in a reflected form (indexed by the original Idris expression) leads to straightforward construction and manipulation of proof objects. I will also show how users (i.e. application programmers) can apply proof procedures without affecting the readability of their programs.

The method I describe is: *lightweight*, in that it requires minimal modification to the Idris type checker and evaluator; *first-class*, in that reflection is implemented by a normal Idris pattern matching definition; and *type-safe* in that the resulting expressions are guaranteed to be faithful representations of the corresponding inputs and any generated proof is guaranteed to be a proof of the required property.

## Example

Consider the following function type:

```
assocP : (x : a) -> (xs, ys : List a) ->
         ((xs ++ (x :: ys ++ xs)) =
            ((xs ++ [x]) ++ (ys ++ xs)))
```

This function represents a proof obligation which can be resolved by repeated application of the following two lemmas:

```
appendNilNeutral : (xs : List a) -> xs ++ [] = xs
appendAssoc : (xs, ys, zs : List a)
              xs ++ (ys ++ zs) = (xs ++ ys) ++ zs
```



To do so by hand for many proof obligations quickly becomes tedious! We can, however, automate such proofs by representing expressions in a reflected form:

```
data Expr : List (List a) -> List a -> Type where
   ENil : Expr G []
   App  : Expr G xs -> Expr G ys -> Expr G (xs ++ ys)
   Var  : Elem xs G -> Expr G xs
```

That is, a list expression is either an empty list `ENil`, representing `[]`, a concatenation of two lists `App`, representing `xs ++ ys`, or an arbitrary list expression `Var`. Since this is a reflected form, we can write functions to manipulate `Expr`s, e.g.

```
reduce : Expr G xs -> (xs' ** (Expr G xs', xs = xs'))
```

That is, given an expression reflecting `xs`, produce a new expression reflecting `xs'` alongside a proof that the new list is equivalent to the original. If `reduce` is written so as to reduce `xs` to a normal form (say, fully right-associative lists) then it is a small step to write the following function which attempts to prove an equality between two lists given their reflected forms:

```
testEq : Expr G xs -> Expr G ys -> Maybe (xs = ys)
```

Similarly, we can reflect equality proofs over list expressions:

```
data ListEq : List (List a) -> Type -> Type where
   EqP : Expr G xs -> Expr G ys -> ListEq G (xs = ys)

tryProof : ListEq G t -> Maybe t
tryProof (EqP xs ys) = testEq xs ys
```

In order to use this in practice, we write reflection functions for lists and equality types, which convert a compile-time list expression (resp. equality type) into the equivalent `Expr`:

```
reflectList : (G : List (List a)) -> (xs : List a) ->
              (G' ** Expr (G' ++ G) xs)
reflectEq : (a : Type) -> (G : List (List a)) ->
            (P : Type) -> (G' ** ListEq (G' ++ G) P)
```

In the talk, I will describe how these reflection functions are implemented, how they are invoked and how they can be combined with `reduce` and `tryProof` above, in such a way that the `assocP` example above can be implemented fully (that is, as a total function) as follows:

```
total
assocP : (x : a) -> (xs, ys : List a) ->
         ((xs ++ (x :: ys ++ xs)) =
            ((xs ++ [x]) ++ (ys ++ xs)))
assocP {a} x xs ys = AssocProof a
```

The result of implementing this reflection machinery is a reusable compile-time decision procedure, `AssocProof`, which either succeeds producing a proof of the required equality, or fails with a compile-time error message.