Epigram Reasoning: Solving Problems With Commutative Monoids

Matthew Walton
mxw@cs.nott.ac.uk

17th January 2007

Epigram

- A dependently-typed functional language by Conor McBride and James McKinna
- Embodies *correct by construction* programming whether you like it or not
- Allows you to consider programs as proofs and proofs as programs

Natural Numbers

- We define the natural numbers in Epigram in a very familiar manner

$$\underline{\text{data}} \quad \left(\frac{}{\mathsf{Nat} \;:\; \star} \right) \quad \underline{\text{where}} \quad \left(\frac{}{\mathsf{zero} \;:\; \mathsf{Nat}} \right) \;;\; \left(\frac{n \;:\; \mathsf{Nat}}{\mathsf{suc}\ n \;:\; \mathsf{Nat}} \right)$$

- Constructors are **zero** and **suc**
- Epigram's syntax resembles natural deduction rules

Vectors

- Vectors are like lists but each vector's length is carried in its type

$$\underline{\text{data}} \left(\frac{n : \mathsf{Nat} \,;\, X : \star}{\mathsf{Vec}\, n\, X : \star}\right) \underline{\text{where}} \left(\frac{x : X}{\mathsf{vnil} : \mathsf{Vec}\,\mathsf{zero}\, X}\right) ; \left(\frac{x : X \,;\, xs : \mathsf{Vec}\, n\, X}{\mathsf{vcons}\, x\, xs : \mathsf{Vec}\,(\mathsf{suc}\, n)\, X}\right)$$

So you want to reverse a vector

- We'll do it with an accumulator

$$\underline{\text{let}} \quad \left(\frac{xs \;:\; \mathsf{Vec}\; m\; X \;;\; ys \;:\; \mathsf{Vec}\; n\; X}{\mathbf{vqrev}\; xs\; ys \;:\; \mathsf{Vec}\; (\mathbf{plus}\; m\; n)\; X} \right)$$

$$
\begin{array}{l}
\mathbf{vqrev}\; xs\; ys \Leftarrow \underline{\text{rec}}\; xs\; \{ \\
\quad \mathbf{vqrev}\; xs\; ys \Leftarrow \underline{\text{case}}\; xs\; \{ \\
\quad\quad \mathbf{vqrev}\; \mathsf{vnil}\; ys \;\Rightarrow\; \mathsf{vnil} \\
\quad\quad \mathbf{vqrev}\; (\mathsf{vcons}\; x\; xs)\; ys\; [] \\
\quad \} \\
\}
\end{array}
$$

Woops

- Unfortunately, it doesn’t work

```
      (   xs : Vec m X ;  ys : Vec n X  !
let   !--------------------------------!
      ! vqrev xs ys : Vec (plus m n) X )

      vqrev xs ys <= rec xs
      { vqrev xs ys <= case xs
        { vqrev vnil ys => ys
          vqrev (vcons x xs) ys => vqrev xs (vcons x ys)
        }
      }
```

What's the problem?

- Although it might not look like it, there is a type mismatch here

Epigram wanted $\mathsf{Vec}\ (\mathsf{suc}\ (\mathbf{plus}\ m\ n))\ X$

We gave it $\mathsf{Vec}\ (\mathbf{plus}\ m\ (\mathsf{suc}\ n))\ X$

- Basic arithmetic tells us the two are equivalent, but Epigram doesn't know that

We're going to have to cheat

- We can define a function called *coerce* which allows type transformations if given a proof that source and target types are equivalent

$$\underline{\text{let}} \quad \left(\frac{q \;:\; S = T}{\mathbf{coerce}\; q \;:\; S \to T} \right)$$

$$\begin{array}{l} \mathbf{coerce}\; q \Leftarrow \underline{\text{case}}\; q\; \{ \\ \quad \mathbf{coerce}\; \mathsf{refl} \;\Rightarrow\; \underline{\text{lam}}\; x \;\Rightarrow\; x \\ \} \end{array}$$

Here's the proof

$$\underline{\text{let}} \left(\frac{m, n \;:\; \mathsf{Nat}}{\mathbf{plusSuc}\; m\; n \;:\; (\mathbf{plus}\; m\; (\mathsf{suc}\; n)) = (\mathsf{suc}\; (\mathbf{plus}\; m\; n))} \right)$$

$$
\begin{array}{l}
\mathbf{plusSuc}\; m\; n \Leftarrow \underline{\text{rec}}\; m\; \{ \\
\quad \mathbf{plusSuc}\; m\; n \Leftarrow \underline{\text{case}}\; m\; \{ \\
\quad\quad \mathbf{plusSuc}\; \mathsf{zero}\; n \;\Rightarrow\; \mathsf{refl} \\
\quad\quad \mathbf{plusSuc}\; (\mathsf{suc}\; m)\; n \;\Rightarrow\; \mathbf{ra}\; (\mathbf{rf}\; \mathsf{suc})\; (\mathbf{plusSuc}\; m\; n) \\
\quad \} \\
\}
\end{array}
$$

One extra required

- plusSuc proves that the vector lengths are equal, but we need to wrap that proof in the vector type in order for it to work in vqrev
- a simple function called vecEq accomplishes this with a minimum of fuss

$$\underline{\text{let}} \quad \left(\frac{q \; : \; S = T}{\mathbf{vecEq}\; q \; : \; \mathsf{Vec}\; S\; X = \mathsf{Vec}\; T\; X} \right) \quad \mathbf{vecEq}\; q \;\Rightarrow\; \mathsf{refl}$$

All fixed

- We have had to add implicit parameters to allow us to talk about the lengths of the vectors

$$\underline{\text{let}}\quad \left(\frac{m;\ n;\ vx\ :\ \mathsf{Vec}\ m\ X\ ;\ ys\ :\ \mathsf{Vec}\ n\ X}{\mathbf{vqrev}\ \_m\ \_n\ xs\ ys\ :\ \mathsf{Vec}\ (\mathbf{plus}\ m\ n)\ X} \right)$$

$$
\begin{array}{l}
\mathbf{vqrev}\ \_m\ \_n\ xs\ ys \Leftarrow \underline{\text{rec}}\ xs\ \{ \\
\quad \mathbf{vqrev}\ \_m\ \_n\ xs\ ys \Leftarrow \underline{\text{case}}\ xs\ \{ \\
\quad\quad \mathbf{vqrev}\ \_\mathsf{zero}\ \_n\ \mathsf{vnil}\ ys\ \Rightarrow\ \mathsf{vnil} \\
\quad\quad \mathbf{vqrev}\ \_(\mathsf{suc}\ m)\ \_n\ (\mathsf{vcons}\ x\ xs)\ ys\ \Rightarrow \\
\quad\quad\quad \mathbf{coerce}\ (\mathbf{vecEq}\ (\mathbf{plusSuc}\ m\ n))\ (\mathbf{vqrev}\ xs\ (\mathsf{vcons}\ x\ ys)) \\
\quad \} \\
\}
\end{array}
$$

What did we use here?

- To resolve a type mismatch we directly proved the equivalence of the crucial parts of the mismatched types
- We then used two type conversion functions – vecEq and coerce – to apply the proof to the problem
- This is fine until we have to do it again for a similar problem

Why do this every time?

- It is tedious to have to define the exact proof required for every single instance of this sort of problem
- It would be much better to have a more flexible way of resolving this kind of simple type conflict
- We need to go and find a pattern

The pattern is found in monoids

- The natural numbers with addition and 0 form a commutative monoid
- The natural numbers also form a commutative monoid with multiplication and 1

Commutative monoids

- A monoid is an algebraic structure consisting of a set of elements, a binary operation and an identity element
- The binary operation is associative, and the identity element is its left and right identity
- A commutative monoid adds the constraint that the binary operation must also be commutative
- Thus the ordering of elements in an expression in a commutative monoid with variables and the binary operation is irrelevant to its meaning

Representing commutative monoids

- We can develop an Epigram data structure which contains the monoid's elements, operation and identity, and the proofs of the necessary laws

...and an expression?

- Representing expressions in a commutative monoid is simple
- Fin, the type of finite sets, is used to represent variables

$$\underline{\text{data}} \left(\frac{n : \mathsf{Nat}}{\mathsf{CMonExp}\ n : \star} \right) \underline{\text{where}} \left(\frac{}{\mathsf{MonZero} : \mathsf{CMonExp}\ n} \right)$$

$$\left(\frac{v : \mathsf{Fin}\ n}{\mathsf{MonVar}\ v : \mathsf{CMonExp}\ n} \right)$$

$$\left(\frac{e, f : \mathsf{CMonExp}\ n}{\mathsf{MonPlus}\ e\ f : \mathsf{MonExp}\ n} \right)$$

Finite sets

- We are using finite sets to represent variables
- They can easily be used to index vectors, which is important to us later on

$$\underline{\text{data}} \; \left(\frac{n \; : \; \mathsf{Nat}}{\mathsf{Fin}\; n \; : \; \star} \right) \; \underline{\text{where}} \; \left(\frac{}{\mathsf{fz} \; : \; \mathsf{Fin}\; (\mathsf{suc}\; n)} \right) \left(\frac{i \; \mathsf{Fin}\; n}{\mathsf{fs}\; i \; : \; \mathsf{Fin}\; (\mathsf{suc}\; n)} \right)$$

Deciding equivalence

- We could use a variety of techniques to apply the monoid laws in an attempt to manipulate the expressions into being equal
- Fortunately there is a much easier way to do things by using normal forms

Multisets: the normal form

- Because we are dealing with a commutative monoid, the ordering of the elements of any expression is irrelevant
- As a consequence, the only thing which actually matters is how many of each element is present
- Therefore we can represent any expression as a normal form which is a multiset of the elements of the expression
- We represent this in Epigram with a vector which records the frequency of each element

Multisets also form a commutative monoid

- The binary operation is multiset sum, represented by element-wise addition of vectors
- The identity element is the empty multiset, represented by the vector of zeroes

Normalisation

- Normalising an expression is easy
- Because multisets are a commutative monoid, we can obtain the normal form of a commutative monoid expression by evaluating the expression much as we would to find its value as an expression of natural numbers, except we use the multiset monoid instead
- During the evaluation, values for variables are taken from the identity matrix

Deciding equality of normal forms

- Two normal forms represented as vectors of multiplicities are equal if and only if the vectors are equal
- Once in normal form, therefore, the expressions may be compared using simple vector equality

Plumbing: how to make it work

- Ideally we would like to be able to write an Epigram expression which takes two expressions and the appropriate monoid data and produces a type conversion function if the expressions are equal
- We're not quite there yet

Plumbing : how to make it work

- Initially we have to produce an expression in a suitable form
- The first step is to convert occurrences of *suc n* to $1 + n$
- All constant values would ideally be added together and then abstracted to a variable, as the expression format does not encode constants
- Different constant values must not yield the same variable

More plumbing

- Once the expressions are represented by *CMonExp* values, we must normalise and compare them
- The result of the comparison needs to be made known to Epigram. It is not difficult to write a function which produces a proof that expressions *e* and *f* are equal iff their normal forms are equal
- This has not been implemented within Epigram 1 as its resource requirements are too high to allow it

Glue

- To do this properly we need glue inside Epigram
- A reflection mechanism which converts expressions to and from *CMonExp* and analagous types for other such problem domains using supplied rules would allow this sort of equality-determination system to be used easily within the code

What's next?

- After everything so far is documented...
- Consider the requirements and nature of a reflection mechanism
- Data types for an Epigram library could come with sets of rules for any suitable structures which they conform to
- Reasoning mechanisms for other sorts of problem would need to be developed
- There are limits to what we can do within the bounds of decidability