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

\[
\text{data } \left( \frac{\text{Nat} : \star}{\text{\fbox{}}} \right) \text{ where } \left( \frac{\text{zero} : \text{Nat}}{\text{\fbox{}}} \right) \text{ ; } \left( \frac{n : \text{Nat}}{\text{\fbox{}}} \right)
\]

- Constructors are \textbf{zero} and \textbf{suc}
- Epigram’s syntax resembles natural deduction rules
Vectors

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

\[
\text{data} \quad \left( \begin{array}{c}
n : \text{Nat} \\
\text{Vec } n \ X : * \\
\end{array} \right) \quad \text{where} \quad \left( \begin{array}{c}
x : X \\
\text{vnil} : \text{Vec } \text{zero } X \\
\text{vcons } x \ xs : \text{Vec } (\text{suc } n) \ X \\
\end{array} \right)
\]
So you want to reverse a vector

- We’ll do it with an accumulator

\[
\text{let } \left(\begin{array}{c}
\text{x}_s : \text{Vec } m \ X \ ; \ \text{y}_s : \text{Vec } n \ X \\
\text{vqrev } \text{x}_s \ \text{y}_s : \text{Vec } (\text{plus } m \ n) \ X
\end{array}\right)
\]

\[
\text{vqrev } \text{x}_s \ \text{y}_s \leftarrow \text{rec } \text{x}_s \ \{ \\
\text{vqrev } \text{x}_s \ \text{y}_s \leftarrow \text{case } \text{x}_s \ \{ \\
\text{vqrev } \text{vnil } \text{y}_s \Rightarrow \text{vnil} \\
\text{vqrev } (\text{vcons } x \ \text{x}_s) \ \text{y}_s \ [\ ]
\}
\}
\]
Woops

- Unfortunately, it doesn't work

```haskell
let (xs : Vec m X ; ys : Vec n X) !
  ! 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 \( \text{Vec} (\text{suc} (\text{plus} \ m \ n)) \ X \)

We gave it \( \text{Vec} (\text{plus} \ m \ (\text{suc} \ n)) \ X \)

- 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

\[
\begin{align*}
\text{let } & \left( \frac{q : S = T}{\text{coerce } q : S \rightarrow T} \right) \\
\text{coerce } q & \leftarrow \text{case } q \begin{cases} \\
\text{coerce } \text{refl} & \Rightarrow \text{lam } x \Rightarrow x \\
\end{cases}
\end{align*}
\]
Here's the proof

\[
\begin{align*}
\text{let } & (m, n : \text{Nat}) \\
\text{plusSuc } m n & : (\text{plus } m (\text{suc } n)) = (\text{suc } (\text{plus } m n))
\end{align*}
\]

\[
\begin{align*}
\text{plusSuc } m n & \leftarrow \text{rec } m \{ \\
\text{plusSuc } m n & \leftarrow \text{case } m \{ \\
\text{plusSuc } \text{zero } n & \Rightarrow \text{refl} \\
\text{plusSuc } (\text{suc } m) n & \Rightarrow \text{ra (rf suc)} (\text{plusSuc } m n)
\}
\}
\}
\]
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

\[
\text{let } \left( \frac{q : S = T}{\text{vecEq } q : \text{Vec } S X = \text{Vec } T X} \right) \text{ vecEq } q \Rightarrow \text{ refl}
\]
• We have had to add implicit parameters to allow us to talk about the lengths of the vectors

\[
\begin{align*}
\text{let} & \quad \left( \begin{array}{c}
m; \ n; \ vx : \ Vec \ m \ X ; \ ys : \ Vec \ n \ X \\
vqrev \ _m \ _n \ xs \ ys : \ Vec \ (\text{plus} \ m \ n) \ X
\end{array} \right) \\
\end{align*}
\]

\[
vqrev \ _m \ _n \ xs \ ys \leftarrow \text{rec} \ xs \ { \\
vqrev \ _m \ _n \ xs \ ys \leftarrow \text{case} \ xs \ { \\
vqrev \ _\text{zero} \ _n \ vnil \ ys \Rightarrow \ vnil \\
vqrev \ _\text{suc} \ m \ _n \ (vcons \ x \ xs) \ ys \Rightarrow \\
\text{coerce} \ (\text{vecEq} \ (\text{plusSuc} \ m \ n)) \ (vqrev \ xs \ (vcons \ x \ ys)) \\
\}}
\]
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

```
data (n : Nat) → CMonExp n : *
where
  MonZero : CMonExp n
  v : Fin n
  MonVar v : CMonExp n
  e, f : CMonExp n
  MonPlus e f : MonExp n```

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

```haskell
data (\n  \n  n : Nat
  \Fin n : \star
) where
  fz : Fin (suc n)
  i \Fin n
  fs i : Fin (suc n)
```
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
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 \textit{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 \texttt{CMonExp} and analogous 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