Generic Programming in a Dependently Typed Language

Generic proofs for generic programs

Peter Morris
pwm@cs.nott.ac.uk

University of Nottingham
This talk

- We introduce a more interesting type for equality testing
  - We return evidence of inequality or inequality
  - this provides proof that the program is correct
  - and allows a more sensible flow of control
This talk

- We introduce a more interesting type for equality testing
  - We return evidence of inequality or inequality
  - this provides proof that the program is correct
  - and allows a more sensible flow of control

- We show how to program with universes for generic programming
  - we introduce the type of regular types
  - and the type of elements for those types
  - so we can write generic programs
This talk

- We introduce a more interesting type for equality testing
  - We return evidence of inequality or inequality
  - this provides proof that the program is correct
  - and allows a more sensible flow of control

- We show how to program with universes for generic programming
  - we introduce the type of regular types
  - and the type of elements for those types
  - so we can write generic programs

- We then combine the two ideas
Nat Equality

- We are used to equality being defined like this:

\[
\begin{align*}
&\text{let } m, n : \text{Nat} \\
&\text{let } \text{eqN} m n : \text{Bool} \\
&\text{eqN} m n \leftarrow \text{rec } n, \text{case } m, \text{case } n \\
&\{ \text{eqN zero zero } \Rightarrow \text{true} \\
&\text{eqN zero (suc } n) \Rightarrow \text{false} \\
&\text{eqN (suc } m) \text{ zero } \Rightarrow \text{false} \\
&\text{eqN (suc } m) \text{ (suc } n) \Rightarrow \text{eqN } m n \\
&\} 
\end{align*}
\]
Nat Equality

- We are used to equality being defined like this:

```plaintext
m, n : Nat
let _________
    eqN m n : Bool

eqN m n <- rec n, case m, case n
{ eqN zero zero => true
  eqN zero (suc n) => false
  eqN (suc m) zero => false
  eqN (suc m) (suc n) => eqN m n
}
```

- But can we do any better with that type?
Nat Equality

- ...wouldn’t THIS be better:

\[
\begin{align*}
\text{let } \ & m, n : \text{Nat} \\
& \text{eqN } m \ n : (m = n) + ((m = n) \rightarrow \emptyset) \\
\text{eqN } m \ n \leftarrow \text{rec } n, \text{case } m, \text{case } n \\
\{ \ & \text{eqN zero zero } \Rightarrow \text{???} \\
& \text{eqN zero (suc } n) \Rightarrow \text{???} \\
& \text{eqN (suc } m) \text{ zero } \Rightarrow \text{???} \\
& \text{eqN (suc } m) \ (\text{suc } n) \Rightarrow \text{eqN } m \ n \text{???} \\
\}
\end{align*}
\]
Nat Equality

- ...wouldn’t THIS be better:

\[
\text{let } m, n : \text{Nat} \\
\text{eqN } m n : (m = n) + ((m = n) \rightarrow \varnothing)
\]

\[
\text{eqN } m n \leftarrow \text{rec } n, \text{case } m, \text{case } n \\
\{ \text{eqN zero zero } \Rightarrow \text{left refl} \\
\text{eqN zero (suc } n) \Rightarrow \text{right } \text{??} \\
\text{eqN (suc } m) \text{ zero } \Rightarrow \text{right } \text{??} \\
\text{eqN (suc } m) (\text{suc } n) \Rightarrow \text{eqN } m n?\text{??} \\
\}\]
Nat Equality

- ...wouldn’t THIS be better:

```
let m, n : Nat
eqN m n : (m = n) + ((m = n) → ∅)

eqN m n ← rec n, case m, case n
{ eqN zero zero ⇒ left refl
  eqN zero (suc n) ⇒ right (λp ← case p)
  eqN (suc m) zero ⇒ right (λq ← case q)
  eqN (suc m) (suc n) ⇒ eqN m n???
}
```

There are no elements of this type so we don’t have to define this function!
Nat Equality

- ...wouldn’t THIS be better:

\[
\begin{aligned}
    \text{let } m, n : \text{Nat} \\
    \text{eqN } m \ n : (m = n) + ((m = n) \rightarrow \emptyset)
\end{aligned}
\]

\[
\begin{aligned}
    \text{eqN } m \ n \leftarrow \text{rec } n, \text{case } m, \text{case } n \\
    \{ \text{eqN zero zero } \Rightarrow \text{left refl} \\
    \quad \text{eqN zero (suc } n) \Rightarrow \text{right } (\lambda p \leftrightarrow \text{case } p) \\
    \quad \text{eqN (suc } m) \text{ zero } \Rightarrow \text{right } (\lambda q \leftrightarrow \text{case } q) \\
    \quad \text{eqN (suc } m) \text{ (suc } n) \Rightarrow \text{eqN } m \ n\\
    \}
\end{aligned}
\]

No! this has the type:

\[(m = n) + ((m = n) \rightarrow \emptyset)\]

we need:

\[(\text{suc } m = \text{suc } n) + ((\text{suc } m = \text{suc } n) \rightarrow \emptyset)\]

so we have to do a bit more work
Nat Equality

• ...wouldn’t THIS be better:

```
let m, n : Nat

eqN m n : (m = n) + ((m = n) → ⊥)

eqN m n ← rec n, case m, case n

{ eqN zero zero ⇒ left refl
  eqN zero (suc n) ⇒ right (λp ← case p)
  eqN (suc m) zero ⇒ right (λq ← case q)
  eqN (suc m) (suc n) || eqN m n

  { eqN (suc m) (suc m) | left refl ⇒ ???
    eqN (suc m) (suc n) | right p ⇒ ???
  }
}
```
Nat Equality

...wouldn’t THIS be better:

\[
\text{let } \quad m, n : \text{Nat} \\
\text{eqN } m \; n \defeq (m = n) + ((m = n) \rightarrow \emptyset) \\
\text{eqN } m \; n \defeq \text{rec } n, \text{case } m, \text{case } n \\
\{ \text{eqN zero zero } \rightarrow \text{ left refl } \\
\quad \text{eqN zero (suc } n \text{) } \rightarrow \text{ right } (\lambda p \leftarrow \text{ case } p) \\
\quad \text{eqN (suc } m \text{) zero } \rightarrow \text{ right } (\lambda q \leftarrow \text{ case } q) \\
\quad \text{eqN (suc } m \text{) (suc } n \text{) } \mid \text{ eqN } m \; n \\
\text{eqN (suc } m \text{) (suc } m \text{)} \mid \text{ left refl } \Rightarrow \text{ left refl } \\
\quad \text{eqN (suc } m \text{) (suc } n \text{)} \mid \text{ right } p \Rightarrow \text{ right } ???
\}
\]
Nat Equality

...wouldn’t THIS be better:

\[
\begin{align*}
\text{let } & \quad m, n : \text{Nat} \\
\text{eqN } m \ n : (m = n) + ((m = n) \rightarrow \emptyset)
\end{align*}
\]

eqN \ m \ n \ \leftarrow \ \text{rec } n, \ \text{case } m, \ \text{case } n

\{ \text{eqN zero zero } \Rightarrow \ \text{left refl} \\
\text{eqN zero (suc } n) \Rightarrow \ \text{right (} \lambda p \leftarrow \text{case } p) \\
\text{eqN (suc } m) \ \text{zero } \Rightarrow \ \text{right (} \lambda q \leftarrow \text{case } q) \\
\text{eqN (suc } m) \ (\text{suc } n) \ || \ \text{eqN } m \ n \\
\{ \ \text{eqN (suc } m) \ (\text{suc } m) \ | \ \text{left refl } \Rightarrow \ \text{left refl} \\
\text{eqN (suc } m) \ (\text{suc } n) \ | \ \text{right } p \Rightarrow \ (\lambda \text{refl } \Rightarrow \ p \ \text{refl}) \\
\} \\
\}

The argument to this function has type:

\text{suc } m = \text{suc } n

if there is such a proof \ m and \ n \ MUST be the same
So what?

- why would we want to do it this way?
So what?

- why would we want to do it this way?
- we don’t just have a program, we have a proof that the program is correct
So what?

- why would we want to do it this way?
- we don’t just have a program, we have a proof that the program is correct
- say we had defined matrix multiplication:
  ```
  x : Matrix l m ; y : Matrix m n
  let matmult x y : Matrix l n
  ```
  but were given two matrices with dimensions \( l \times m \) and \( n \times o \)?
So what?

- why would we want to do it this way?
- we don’t just have a program, we have a proof that the program is correct
- say we had defined matrix multiplication:
  \[
  x : \text{Matrix } l \times m ;
  y : \text{Matrix } m \times n
  \]
  let \[
  \text{matmult } x y : \text{Matrix } l \times n
  \]
  but were given two matrices with dimensions \( l \times m \) and \( n \times o \)?
- a proof that \((n = \text{zero}) \rightarrow \emptyset\) is exactly what we need to write \( m \div n \) by structural recursion
So what?

- Why would we want to do it this way?

- We don’t just have a program, we have a proof that the program is correct.

- Say we had defined matrix multiplication:

  ```
  x : Matrix l m ; y : Matrix m n
  let matmult x y : Matrix l n
  ```

  But were given two matrices with dimensions \( l \times m \) and \( n \times o \)?

- A proof that \( ((n = \text{zero}) \rightarrow \emptyset) \) is exactly what we need to write \( m \text{ `div` } n \) by structural recursion.

- Shift your view of programming - obviously correct, elegant programs.
In generic programming we try and define programs that work for a range of data types, to prevent reproduction of code.
In generic programming we try and define programs that work for a range of data types, to prevent reproduction of code.

One class of types is the Regular Data types which are defined using:

\[ \mu, +, \times, 1 \text{ and } 0 \]
In generic programming we try and define programs that work for a range of data types, to prevent reproduction of code.

One class of types is the Regular Data types which are defined using:

\[ \mu, +, \times, \mathbf{1} \text{ and } \mathbf{0} \]

For instance:

\[
\begin{align*}
\text{Nat} &= \mu X. \mathbf{1} + X \\
\text{List A} &= \mu Z. \mathbf{1} + (A \times Z) \\
\text{BTree B} &= \mu Y. B + (Y \times Y)
\end{align*}
\]
...in Epigram

- We can define the Epigram type of Regular Types in n variables (we use debruijn indicies instead of the names above)
...in Epigram

- We can define the Epigram type of Regular Types in \( n \) variables (we use debruijn indices instead of the names above)

\[
\begin{align*}
\text{n : Nat} \\
\text{data \_\_\_\_\_\_ where \_\_\_\_\_\_; \_\_\_\_\_\_;} \\
\text{RegType n : \star} \hspace{1em} \text{Zero : RegType n} \hspace{1em} \text{One : RegType n} \\
\text{\_\_\_\_\_\_; \_\_\_\_\_;} \\
\text{Union l r : RegType n} \hspace{1em} \text{Product x y : RegType n} \\
\text{\_\_\_\_\_\_; \_\_\_\_\_;} \\
\text{t : RegType (suc n)} \hspace{1em} \text{rts t : RegType n} \hspace{1em} \text{rtz : RegType (suc n)} \hspace{1em} \text{Mu t : RegType n}
\end{align*}
\]
We can then define the type of elements of a given closed Regular Type:
We can then define the type of elements of a given closed Regular Type:

\[ rt : \text{RegType} \]

\[
data \begin{array}{l}
    \text{Elem } rt : \star \\
    \text{ea} : \text{Elem } rta ; \text{eb} : \text{Elem } rtb \\
    \text{unit} : \text{Elem } \text{One} ; \text{pair } ea \; eb : \text{Elem } (\text{Product } rta \; rtb) \\
    \text{inl } ea : \text{Elem } (\text{Union } rta \; rtb) ; \text{inr } eb : \text{Elem } (\text{Union } rta \; rtb) \\
    e : \text{Elem'} (\varepsilon, (\text{Mu } f)) \; f \\
    \text{in } e : \text{Elem } (\text{Mu } f)
\end{array}
\]
We can then define the type of elements of a given closed Regular Type:

\[ rt : \text{RegType} \quad \text{zero} \]

\[
\text{data} \quad \text{Elem} \, rt : * \quad \text{where}
\]

\[
\begin{align*}
\text{elem} : \text{Elem} \, rt & : \star \\
\text{ea} : \text{Elem} \, rta & ; \text{eb} : \text{Elem} \, rtb \\
\text{unit} : \text{Elem} \, \text{One} & \quad \text{pair} \, \text{ea} \, \text{eb} : \text{Elem} \, (\text{Product} \, \text{rta} \, \text{rtb})
\end{align*}
\]

\[
\begin{align*}
\text{inl} \, \text{ea} : \text{Elem} \, (\text{Union} \, \text{rta} \, \text{rtb}) & ; \quad \text{inr} \, \text{eb} : \text{Elem} \, (\text{Union} \, \text{rta} \, \text{rtb})
\end{align*}
\]

\[
\begin{align*}
\text{e} : \text{Elem}' \, (\varepsilon, (\text{Mu} \, f)) & \quad \text{f} \\
\text{in} \, \text{e} : \text{Elem} \, (\text{Mu} \, f)
\end{align*}
\]

We actually need the more general \text{Elem'} over \text{RegTypes} of a given context (see the \text{Mu} case!).
Reflection

- What can we do with this? Well, we reflect real Epigram types into our representation. So for instance the judgment:

\[
suc \ (suc \ zero) : \mathbb{N}
\]
Reflection

- What can we do with this? Well, we reflect real Epigram types into our representation. So for instance the judgment:
  \[ \text{suc} \ (\text{suc} \ \text{zero}) : \mathbb{N} \]
- can be reflected in to the judgment:
  \[ \text{in} \ (\text{inr} \ (\text{last} \ (\text{in} \ (\text{inr} \ \text{last} \ (\text{in} \ (\text{inl} \ \text{unit})))))))) : \text{Elem} \ (\text{Mu} \ (\text{Union One rtz})) \]
• What can we do with this? Well, we reflect real Epigram types into our representation. So for instance the judgment:

\[
suc \ (suc \ zero) : \mathbb{N}
\]

• can be reflected into the judgment:

\[
in \ (\text{inr} \ (\text{last} \ (\text{inr} \ (\text{last} \ (\text{in} \ (\text{inr} \ (\text{last} \ (\text{in} \ (\text{inl} \ \text{unit}))))))))) : \text{Elem} \ (\text{Mu} \ (\text{Union} \ \text{One} \ \text{rtz}))
\]

• Define programs that are generic over their type by providing its representation as an argument. For instance Generic Equality:

\[
\text{let } rt : \text{RegType} \ \text{zero} ; \ x, y : \text{Elem} \ rt
\]

\[
\text{let eqG} \ x \ y : \text{Bool}
\]
Generic Equality

- Here is the code for the general case over n variables:

  \[
  \text{eqG } \Gamma \ x \ y \leftarrow \text{rec } x, \text{case } x, \text{case } y \\
  \{ \text{eqG } \Gamma \ \text{unit unit } \Rightarrow \text{true} \\
  \{ \text{eqG } \Gamma \ (\text{pair } xa \ xb) \ (\text{pair } ya \ yb) \\
  \Rightarrow \text{eqG } \Gamma \ xa \ ya \ \land \ \text{eqG } \Gamma \ xb \ yb \\
  \text{eqG } \Gamma \ (\text{inl } xa) \ (\text{inl } ya) \Rightarrow \text{eqG } \Gamma \ xa \ ya \\
  \text{eqG } \Gamma \ (\text{inl } xa) \ (\text{inr } yb) \Rightarrow \text{false} \\
  \text{eqG } \Gamma \ (\text{inr } xb) \ (\text{inl } ya) \Rightarrow \text{false} \\
  \text{eqG } \Gamma \ (\text{inr } xb) \ (\text{inr } yb) \Rightarrow \text{eqG } \Gamma \ xb \ yb \\
  \text{eqG}_{\text{Mu } f} \ (\text{in } x) \ (\text{in } y) \Rightarrow \text{eqG } (\Gamma, (\text{Mu } f)) \ x \ y \\
  \text{eqG } (\Gamma, rt') \ (\text{wk } x) \ (\text{wk } y) \Rightarrow \text{eqG } \Gamma \ x \ y \\
  \text{eqG } (\Gamma, rt) \ (\text{last } x) \ (\text{last } y) \Rightarrow \text{eqG } \Gamma \ x \ y \\
  \}\}
  \}
Here is the code for the general case over n variables:

\[
\text{eqG } \Gamma x y \iff \text{rec } x, \text{case } x, \text{case } y
\]

\[
\begin{align*}
\{ & \text{eqG } \Gamma \text{ unit unit } \Rightarrow \text{true} \\
& \{ \text{eqG } \Gamma (\text{pair } xa \ xb) (\text{pair } ya \ yb) \\
& \quad \Rightarrow \text{eqG } \Gamma xa ya \land \text{eqG } \Gamma xb yb \\
& \quad \text{eqG } \Gamma (\text{inl } xa) (\text{inl } ya) \Rightarrow \text{eqG } \Gamma xa ya \\
& \quad \text{eqG } \Gamma (\text{inl } xa) (\text{inr } yb) \Rightarrow \text{false} \\
& \quad \text{eqG } \Gamma (\text{inr } xb) (\text{inl } ya) \Rightarrow \text{false} \\
& \quad \text{eqG } \Gamma (\text{inr } xb) (\text{inr } yb) \Rightarrow \text{eqG } \Gamma xb yb \\
& \quad \text{eqG}_{(\text{Mu } f)} (\text{in } x) (\text{in } y) \Rightarrow \text{eqG } (\Gamma, (\text{Mu } f)) x y \\
& \quad \text{eqG } (\Gamma, rt') (\text{wk } x) (\text{wk } y) \Rightarrow \text{eqG } \Gamma x y \\
& \quad \text{eqG } (\Gamma, rt) (\text{last } x) (\text{last } y) \Rightarrow \text{eqG } \Gamma x y \\
\} \\
\}
\]

But can we do any better with the type?
Yes! We have a generic equality test with this type:

\[
\text{let } \quad \begin{array}{c}
rt : \text{RegType } \text{zero} ;
\end{array} \quad \begin{array}{c}
x, y : \text{Elem } rt
\end{array}
\]

\[
\text{eqG } rt \quad x \quad y : (x = y) + ((x = y) \rightarrow \emptyset)
\]
Yes! We have a generic equality test with this type:

\[
\text{let } \quad \mathit{rt} : \text{RegType } \text{zero} ; \ x, y : \text{Elem } \mathit{rt} \\
\text{eqG } \mathit{rt} \ x \ y : (x = y) + ((x = y) \to \emptyset)
\]

That is we have an equality test that
Yes! We have a generic equality test with this type:

\[
\text{let } \begin{array}{l}
rt : \text{RegType} \ \text{zero} \ ; \ x, y : \text{Elem} \ rt \\
\text{eqG } rt \ x \ y : (x = y) + ((x = y) \rightarrow \emptyset)
\end{array}
\]

That is we have an equality test that

- is generic over all regular types
Yes! We have a generic equality test with this type:

\[
\text{let } rt : \text{RegType } \text{zero} ; x, y : \text{Elem } rt \\
\text{eqG } rt x y : (x = y) + ((x = y) \rightarrow \emptyset)
\]

That is we have an equality test that

- is generic over all regular types
- provides generic evidence of equality or inequality
Yes! We have a generic equality test with this type:

\[
\text{let } \quad rt : \text{RegType} \text{ zero} ; \quad x, y : \text{Elem} \ rt
\]

\[
\text{eqG } rt \ x \ y : (x = y) + ((x = y) \rightarrow \emptyset)
\]

That is we have an equality test that
- is generic over all regular types
- provides generic evidence of equality or inequality
- and is its own proofs of correctness
Yes! We have a generic equality test with this type:

\[
\text{let } rt : \text{RegType zero} ; x, y : \text{Elem } rt \\
\text{eqG } rt x y : (x = y) + ((x = y) \rightarrow \emptyset)
\]

That is we have an equality test that
  - is generic over all regular types
  - provides generic evidence of equality or inequality
  - and is its own proofs of correctness
  - it’s also a proof of the decidability of equivalence for regular types!
Next Step

- We could extend our universe to include all strictly positive types:

\[ \mu, \nu, +, \times, K \rightarrow \]
Next Step

- We could extend our universe to include all strictly positive types:
  \[ \mu, \nu, +, \times, K \rightarrow \]

- But then we lose the ability to write our equality function...
Next Step

- We could extend our universe to include all strictly positive types:

\[ \mu, \nu, +, \times, K \rightarrow \]

- But then we lose the ability to write our equality function...

- We need a number of universes, one for each class of types, regular types and strictly positive types being just two
Next Step

- We could extend our universe to include all strictly positive types:

\[ \mu, \nu, +, \times, K \rightarrow \]

- But then we lose the ability to write our equality function...

- We need a number of universes, one for each class of types, regular types and strictly positive types being just two

- Of course each regular type is strictly positive and so on
The future

- Write more generic programs
  - Ordering for regular types
  - Derivatives of types
The future

- Write more generic programs
  - Ordering for regular types
  - Derivatives of types

- Build more universes
  - Ultimate goal: reflect all* Epigram types in Epigram

*All refers to the Epigram types in the context of the discussion.
The future

- Write more generic programs
  - Ordering for regular types
  - Derivatives of types

- Build more universes
  - Ultimate goal: reflect all* Epigram types in Epigram

- Investigate connection to Containers
  - To provide a categorical theory
The future

- Write more generic programs
  - Ordering for regular types
  - Derivatives of types

- Build more universes
  - Ultimate goal: reflect all* Epigram types in Epigram

- Investigate connection to Containers
  - To provide a categorical theory

- Thank you for listening