



# Containing Families

## *An introduction to Containers*

Peter Morris

w/ Thorsten Altenkirch

`pwm@cs.nott.ac.uk`

University of Nottingham

# A Problem...



- Find a theory of datatypes that supports reusable libraries of code
  - Boilerplate
  - Generics — Haskell deriving



# A Problem...

- Find a theory of datatypes that supports reusable libraries of code
  - Boilerplate
  - Generics — Haskell deriving
- Some types might support
  - Equality
  - Traversability



# A Problem...

- Find a theory of datatypes that supports reusable libraries of code
  - Boilerplate
  - Generics — Haskell deriving
- Some types might support
  - Equality
  - Traversability
- Should be intuitive to use

# An opportunity...



- Epigram a new dependently typed functional programming language



# An opportunity...

- Epigram a new dependently typed functional programming language
- It needs to have reusable (generic) libraries
  - Map for Lists
  - Map for Vectors
  - Map for Telescopes



# An opportunity...

- Epigram a new dependently typed functional programming language
- It needs to have reusable (generic) libraries
  - Map for Lists
  - Map for Vectors
  - Map for Telescopes
- Its notion of datatype is up for grabs



# An opportunity...

- Epigram a new dependently typed functional programming language
- It needs to have reusable (generic) libraries
  - Map for Lists
  - Map for Vectors
  - Map for Telescopes
- Its notion of datatype is up for grabs
- Vectors:

$$\text{data } \frac{n : \text{Nat} ; A : \star}{\text{Vec } n A : \star} \text{ where } \frac{}{\text{nil} : \text{Vec } 0 A} ; \frac{a : A ; as : \text{Vec } n A}{\text{cons } a as : \text{Vec } (1 + n) A}$$



# Positivity (Luo)

- In the current version we use Luo's syntactic definition of strictly positive inductive families. This doesn't allow us to have:

$$\frac{f : (X \rightarrow \text{Bool}) \rightarrow \text{Bool}}{c f : X}$$



# Positivity (Luo)

- In the current version we use Luo's syntactic definition of strictly positive inductive families. This doesn't allow us to have:

$$\frac{f : (X \rightarrow \text{Bool}) \rightarrow \text{Bool}}{c f : X}$$

- Why would we want this anyway?



# Positivity (Luo)

- In the current version we use Luo's syntactic definition of strictly positive inductive families. This doesn't allow us to have:

$$\frac{f : (X \rightarrow \text{Bool}) \rightarrow \text{Bool}}{c f : X}$$

- Why would we want this anyway?
- The type we are defining can only appear to the right of arrows:

$$\frac{f : \text{Bool} \rightarrow X}{c f : X}$$



# But hang on...

- What about definitions like the following:

$$\frac{ts : \text{List} (\text{RoseTree } A)}{\text{node } ts : \text{RoseTree } A}$$



# But hang on...

- What about definitions like the following:

$$\frac{ts : \text{List } (\text{RoseTree } A)}{\text{node } ts : \text{RoseTree } A}$$

- This is forbidden; the syntactic test can't know that `List` preserves positivity

# But hang on...

- What about definitions like the following:

$$\frac{ts : \text{List } (\text{RoseTree } A)}{\text{node } ts : \text{RoseTree } A}$$

- This is forbidden; the syntactic test can't know that `List` preserves positivity
- We can have an encoding of Rose Trees, but they won't be the ones we want!
  - The definition above is the most concise and intuitive
  - We want to use the real `List` functions to map across sub-nodes



# What are containers?

Containers have a set of *Shapes*.  $S : \star \dots$



# What are containers?

Containers have a set of **Shapes**.  $S : \star \dots$

For example:

Lists have a shape very similar to the natural numbers...





# What are containers?

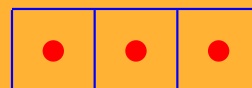
Containers have a set of **Shapes**.  $S : \star \dots$

and for each shape, some set of **Positions** where data goes

$P : \forall s : S \Rightarrow \star$

For example:

Lists have a shape very similar to the natural numbers... and given a shape (say 3) the set of positions has exactly that many elements





# What are containers?

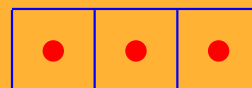
Containers have a set of **Shapes**.  $S : \star \dots$

and for each shape, some set of **Positions** where data goes

$P : \forall s : S \Rightarrow \star$

For example:

Lists have a shape very similar to the natural numbers... and given a shape (say 3) the set of positions has exactly that many elements



Trees: The shape of Binary trees with data at the leaves is a Binary Tree with no room for data, Positions are then paths to leaves...



# More Containers

- Closed under:

$$\mu, +, \times, K \rightarrow$$

... Strictly Positive Types



# More Containers

- Closed under:

$$\mu, +, \times, K \rightarrow$$

... Strictly Positive Types

- Elements of Container types — pick a shape, and for all the positions give a piece of payload

# More Containers

- Closed under:

$$\mu, +, \times, K \rightarrow$$

... Strictly Positive Types

- Elements of Container types — pick a shape, and for all the positions give a piece of payload
- Or define codes for types  $(\mu, +, \times \dots)$  and for each code a type of elements
  - Generic Equality, Map, Differentiation...
  - See previous BCTCS talk



# Indexed Containers

- For each shape, an input sort from  $I$  : ★



# Indexed Containers

- For each shape, an input sort from  $I : \star$
- For each position an output sort from  $O : \forall s : S \Rightarrow \star$



# Indexed Containers

- For each shape, an input sort from  $I : \star$
- For each position an output sort from  $O : \forall s : S \Rightarrow \star$
- For example, Vectors are indexed by the natural numbers, we take lists and copy the shape across to the input sort



# Indexed Containers

- For each shape, an input sort from  $I : \star$
- For each position an output sort from  $O : \forall s : S \Rightarrow \star$
- For example, Vectors are indexed by the natural numbers, we take lists and copy the shape across to the input sort

$$\frac{I, O : \star}{\text{IC } I \ O : \star} \quad \text{where} \quad \frac{S : O \rightarrow \star \quad P : \forall o : O; s : S \ o \Rightarrow I \rightarrow \star}{S \triangleleft P : \text{IC } I \ O}$$



# Roses again

- We can compose containers  
Given a container, with input index type  $J$  and output  $O$ , and a sub-container, with inputs  $I$  and outputs  $J$ , their composition is indexed by  $I$  and  $O$ .



# Roses again

- We can compose containers  
Given a container, with input index type  $J$  and output  $O$ , and a sub-container, with inputs  $I$  and outputs  $J$ , their composition is indexed by  $I$  and  $O$ .
- Since List is a container, and ICs are closed under fixpoints, we can have the previous definition of `RoseTree`.

# Roses again

- We can compose containers  
Given a container, with input index type  $J$  and output  $O$ , and a sub-container, with inputs  $I$  and outputs  $J$ , their composition is indexed by  $I$  and  $O$ .
- Since List is a container, and ICs are closed under fixpoints, we can have the previous definition of `RoseTree`.
- More generally, we have a compositional *semantic* notion of strict positivity for datatypes.

# ICs and Generics

- We define an Epigram datatype which gives a syntax for ICs:

$$\mu, \Pi, \Sigma, \Delta, K \rightarrow \dots$$

# ICs and Generics

- We define an Epigram datatype which gives a syntax for ICs:

$$\mu, \Pi, \Sigma, \Delta, K \rightarrow \dots$$

- ... along with an interpretation, that given a code builds a type that is isomorphic to the type the code represents. This is also an Epigram datatype.

# ICs and Generics

- We define an Epigram datatype which gives a syntax for ICs:

$$\mu, \Pi, \Sigma, \Delta, K \rightarrow \dots$$

- ... along with an interpretation, that given a code builds a type that is isomorphic to the type the code represents. This is also an Epigram datatype.
- If we write programs parametrised by these codes and interpretations we have generic programs.
  - Generic Equality, Map, Modalities?, Differentiation
  - Composition + generics = reusable libraries for Epigram

# SPF



$$\text{data } \frac{\vec{I} : \text{Vec } \star \ n \quad O : \star}{\text{SPF } \vec{I} \ O : \star}$$

$$\text{where } \frac{}{\text{'Z'} : \text{SPF } (\vec{I} :: O) \ O} \quad \frac{T : \text{SPF } \vec{I} \ O}{\text{'wk'} \ T : \text{SPF } (\vec{I} :: I) \ O}$$

$$\frac{f : \forall t : \text{Fin } n \Rightarrow \text{SPF } \vec{I} \ O}{\text{'Tag'} \ f : \text{SPF } \vec{I} \ (O \times \text{Fin } n)} \quad \frac{}{\text{'0'}, \text{'1'} : \text{SPF } \vec{I} \ O}$$

$$\frac{f : O \rightarrow O' \quad T : \text{SPF } \vec{I} \ O}{\text{'\Sigma'} \ O \ f \ T : \text{SPF } \vec{I} \ O'} \quad \frac{f : O' \rightarrow O \quad T : \text{SPF } \vec{I} \ O}{\text{'\Delta'} \ O \ f \ T : \text{SPF } \vec{I} \ O'}$$

$$\frac{f : O \rightarrow O' \quad T : \text{SPF } \vec{I} \ O}{\text{'\Pi'} \ O \ f \ T : \text{SPF } \vec{I} \ O'} \quad \frac{T : \text{SPF } (\vec{I} :: O) \ O}{\text{'\mu'} \ T : \text{SPF } \vec{I} \ O}$$

# Epigram eats itself



- Our syntax can represent all strictly positive inductive families.



# Epigram eats itself

- Our syntax can represent all strictly positive inductive families.
- It is *itself* a strictly positive inductive family.



# Epigram eats itself

- Our syntax can represent all strictly positive inductive families.
- It is *itself* a strictly positive inductive family.
- We can give the code for the code, modulo universe sizes.  
Epigram in Epigram?



# Epigram eats itself

- Our syntax can represent all strictly positive inductive families.
- It is *itself* a strictly positive inductive family.
- We can give the code for the code, modulo universe sizes.  
Epigram in Epigram?
- Strong theory of containers (Abbott, Altenkirch, Ghani, Hancock, McBride) can be used to justify these constructs.



# Future directions

- SPFs are a large universe of types
  - The larger the universe, the less structure, the fewer generic programs it can support
  - Our equality already uses Regular Families ( $\omega$ -continuous functors) because we need to know there are only finitely many positions
  - Traversability is one thing that active container research is focusing on



# Future directions

- SPFs are a large universe of types
  - The larger the universe, the less structure, the fewer generic programs it can support
  - Our equality already uses Regular Families ( $\omega$ -continuous functors) because we need to know there are only finitely many positions
  - Traversability is one thing that active container research is focusing on
- Work in progress
  - Connection to high level syntax
  - How to make writing generic programs as easy as possible within the language
  - Actually implement this stuff in Epigram2

# The End



- More information:
  - Containers: <http://sneezy.cs.nott.ac.uk/containers>
  - Epigram: <http://www.e-pig.org>
- Thanks for your attention!