Dependent Containers

Thorsten Altenkirch
based on joint work with Neil Ghani, Conor McBride and Peter Hancock
University of Nottingham

What is a container?

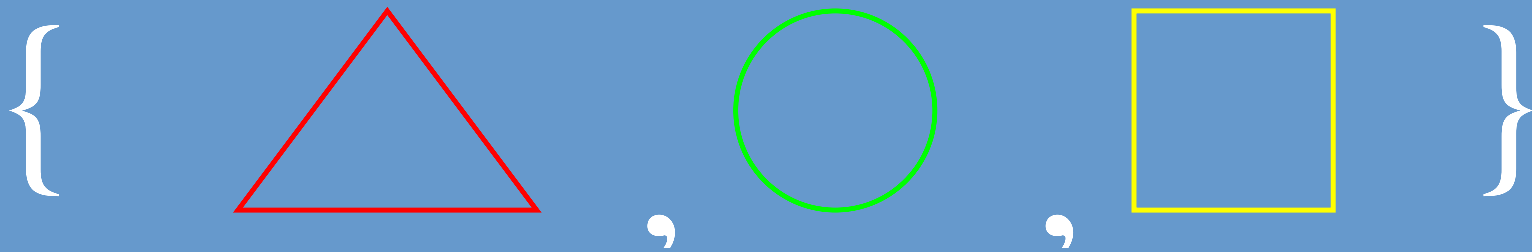
What is a container?

A container type $S \triangleright P$ is given by:

What is a container?

A container type $S \triangleright P$ is given by:

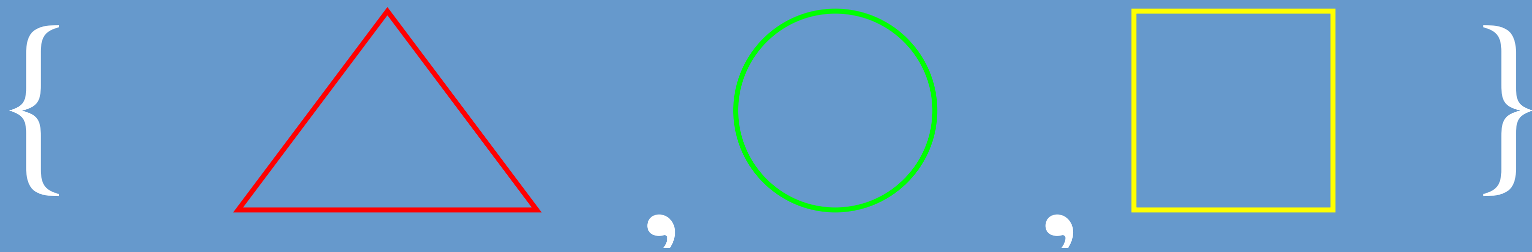
- A set S of shapes, e.g.



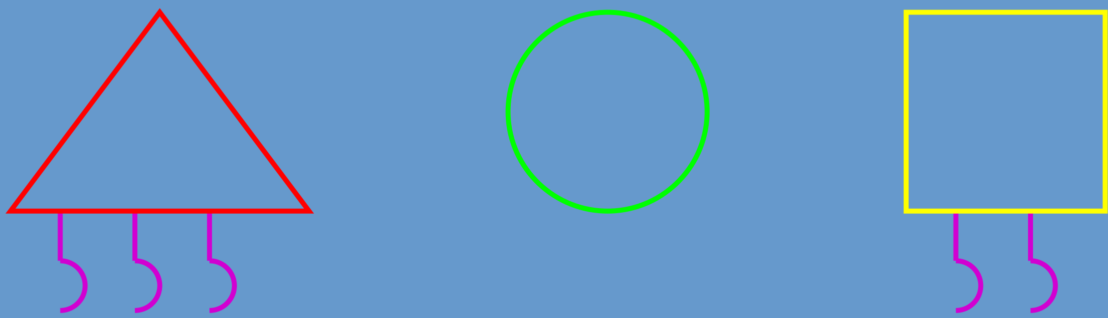
What is a container?

A container type $S \triangleright P$ is given by:

- A set S of shapes, e.g.



- For any shape $s \in S$ a set of positions $P(s)$, e.g.



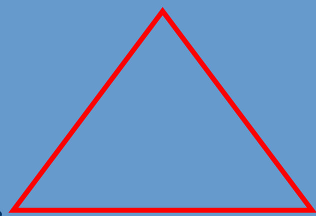
What to do with a container?

What to do with a container?

Given some payload X, e.g. $X = \mathsf{Nat}$ we can instantiate a container by

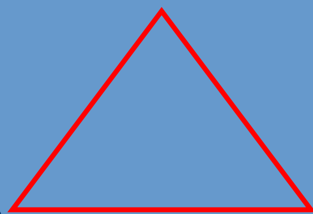
What to do with a container?

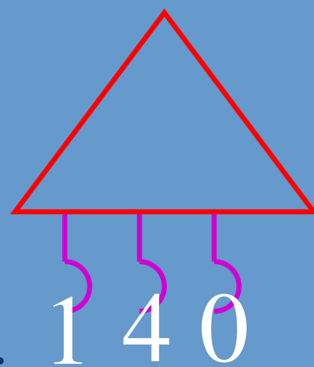
Given some payload X, e.g. $X = \mathsf{Nat}$ we can instantiate a container by

- Choosing a shape, e.g. 

What to do with a container?

Given some payload X, e.g. $X = \mathsf{Nat}$ we can instantiate a container by

- Choosing a shape, e.g. 

- Filling the positions with payload, e.g. e.g. 

Extension of a container type

Extension of a container type

The extension $[\![S \triangleright P]\!]$ of a container is given by an endofunctor $\mathbf{Set} \to \mathbf{Set}$:

$$[\![S \triangleright P]\!](X) = \Sigma s \in S.P(s) \to X$$

Extension of a container type

The extension $[\![S \triangleright P]\!]$ of a container is given by an endofunctor $\mathbf{Set} \to \mathbf{Set}$:

$$[\![S \;\triangleright\; P]\!](X) = \Sigma s \in S.P(s) \to X$$

where

$$\Sigma a \in A.B(a) = \{(a, b) \mid a \in A \wedge b \in B(a)\}$$

Example of a container type

Example of a container type

$$\underline{\text{data}}\ \frac{X : \star}{\mathsf{List}\ X : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{nil} : \mathsf{List}\ X}\quad \frac{x : X \quad xs : \mathsf{List}\ X}{x :: xs : \mathsf{List}\ X}$$

Example of a container type

$$\underline{\text{data}}\ \frac{X : \star}{\mathsf{List}\ X : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{nil} : \mathsf{List}\ X}\ \ \frac{x : X \quad xs : \mathsf{List}\ X}{x :: xs : \mathsf{List}\ X}$$

$$\mathsf{List}\ X = \mu Y.1 + X \times Y$$

Example of a container type

$$\underline{\text{data}}\ \frac{X : \star}{\mathsf{List}\ X : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{nil} : \mathsf{List}\ X}\quad \frac{x : X \quad xs : \mathsf{List}\ X}{x :: xs : \mathsf{List}\ X}$$

$$\mathsf{List}\, X = \mu Y.1 + X \times Y$$

$$\mathsf{List}\, X \quad \simeq \quad \Sigma n \in \mathsf{Nat}.\{i < n\} \to X$$

Example of a container type

$$\underline{\text{data}}\ \frac{X : \star}{\mathsf{List}\ X : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{nil} : \mathsf{List}\ X}\quad \frac{x : X \quad xs : \mathsf{List}\ X}{x :: xs : \mathsf{List}\ X}$$

$$\mathsf{List}\, X = \mu Y.1 + X \times Y$$

$$\begin{aligned} \mathsf{List}\, X &\simeq \Sigma n \in \mathsf{Nat}.\{i < n\} \to X \\ &= \Sigma n \in \mathsf{Nat}.n \to X \end{aligned}$$

n-ary containers

n-ary containers

An n-ary container $S \triangleright \vec{P}$ is given by

- $S \; : \; \mathbf{Set}$
- $\vec{P} \; : \; n \to S \to \mathbf{Set}$

n-ary containers

An n-ary container $S \triangleright \vec{P}$ is given by

- $S \; : \; \mathbf{Set}$
- $\vec{P} \; : \; n \to S \to \mathbf{Set}$

Its extension is an endofunctor $\mathbf{Set}^n \to \mathbf{Set}$ is:

$$[\![S \;\triangleright\; \vec{P}]\!](X) = \Sigma s \in S.\Pi i < n.P\, i\, s \to X\, i$$

Morphisms of containers

Morphisms of containers

Given containers

$$\begin{aligned} F(X) &= \Sigma s \in S.P(s) \to X \\ G(X) &= \Sigma t \in T.Q(t) \to X \end{aligned}$$

a morphism $f \triangleright u \in \mathbf{Con}(F, G)$ is given by

Morphisms of containers

Given containers

$$\begin{aligned} F(X) &= \Sigma s \in S.P(s) \to X \\ G(X) &= \Sigma t \in T.Q(t) \to X \end{aligned}$$

a morphism $f \triangleright u \in \mathbf{Con}(F, G)$ is given by

$$f \in S \to T$$

Morphisms of containers

Given containers

$$\begin{aligned} F(X) &= \Sigma s \in S.P(s) \to X \\ G(X) &= \Sigma t \in T.Q(t) \to X \end{aligned}$$

a morphism $f \triangleright u \in \mathbf{Con}(F, G)$ is given by

$$\begin{aligned} f &\in S \to T \\ u &\in \Pi s \in S.Q(f(s)) \to P(s) \end{aligned}$$

Morphisms of containers

Given containers

$$\begin{aligned} F(X) &= \Sigma s \in S.P(s) \to X \\ G(X) &= \Sigma t \in T.Q(t) \to X \end{aligned}$$

a morphism $f \triangleright u \in \mathbf{Con}(F, G)$ is given by

$$\begin{aligned} f &\in S \to T \\ u &\in \Pi s \in S.Q(f(s)) \to P(s) \\ [\![f \triangleright u]\!]_X &\in F(X) \to G(X) \end{aligned}$$

Morphisms of containers

Given containers

$$\begin{aligned} F(X) &= \Sigma s \in S.P(s) \to X \\ G(X) &= \Sigma t \in T.Q(t) \to X \end{aligned}$$

a morphism $f \triangleright u \in \mathbf{Con}(F, G)$ is given by

$$\begin{aligned} f &\in S \to T \\ u &\in \Pi s \in S.Q(f(s)) \to P(s) \\ [\![f \triangleright u]\!]_X &\in F(X) \to G(X) \\ &= (s, h) \mapsto (f(s), h \circ u(s)) \end{aligned}$$

Representation theorem

Representation theorem

Theorem : Every natural transformation (i.e. polymorphic function) between containers can be represented as a container morphisms.
$[\![\,]\!]$ is full and faithful.

Representation theorem

Theorem : Every natural transformation (i.e. polymorphic function) between containers can be represented as a container morphisms.
$[\![\,]\!]$ is full and faithful.

Example: any natural transformation $g \in \Pi X.\text{List}\,X \to \text{List}\,X$ is given by:

$$\begin{aligned} f &: \mathsf{Nat} \to \mathsf{Nat} \\ u &: \Pi n : \mathsf{Nat}.(f\,n) \to n \end{aligned}$$

Strictly positive types

Strictly positive types

- **Strictly positive types** are generated by 0,1,$+$,$\times$,$C \to -$ (constant exponentation), μ (initial algebra) and ν (terminal coalgebra).

Strictly positive types

- **Strictly positive types** are generated by $0,1,+,\times,C \to -$ (constant exponentation), μ (initial algebra) and ν (terminal coalgebra).
- A **Martin-Löf category** is an extensive, locally cartesian closed category with W-types.

Strictly positive types

- **Strictly positive types** are generated by $0, 1, +, \times, C \rightarrow -$ (constant exponentation), μ (initial algebra) and ν (terminal coalgebra).
- A **Martin-Löf category** is an extensive, locally cartesian closed category with W-types.
- **Theorem:** All strictly positive types are representable as containers in any Martin-Löf category.

Strictly positive types

- **Strictly positive types** are generated by $0,1,+,\times,C \to -$ (constant exponentation), μ (initial algebra) and ν (terminal coalgebra).
- A **Martin-Löf category** is an extensive, locally cartesian closed category with W-types.
- **Theorem:** All strictly positive types are representable as containers in any Martin-Löf category.
- **Corollary :** All closed strictly positive types are representable in any Martin-Löf category.

Observations

Observations

- Framework to define and reason about datatype generic programming, e.g. see our papers on derivatives of containers.

Observations

- Framework to define and reason about datatype generic programming, e.g. see our papers on derivatives of containers.
- Martin-Löf categories have representations of all strictly positive **non-dependent** inductive and coinductive types.

Observations

- Framework to define and reason about datatype generic programming, e.g. see our papers on derivatives of containers.
- Martin-Löf categories have representations of all strictly positive **non-dependent** inductive and coinductive types.
- We have developed a theory of **non-dependent** datatypes in a **dependently** typed framework.

Dependently typed programming

Dependently typed programming

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat} \quad X : \star}{\mathsf{Vec}\ n\ X : \star}\ \underline{\text{where}}\ \frac{}{\overline{\mathsf{nil}} : \mathsf{Vec}\ 0\ X}\quad \frac{x : X \quad xs : \mathsf{Vec}\ n\ X}{x \mathbin{\bar{::}} xs : \mathsf{Vec}\ (1{+}n)\ X}$$

Dependently typed programming

$$\underline{\text{data}}\ \ \frac{n : \mathsf{Nat} \quad X : \star}{\mathsf{Vec}\ n\ X : \star}\ \ \underline{\text{where}}\ \ \frac{}{\overline{\mathsf{nil}} : \mathsf{Vec}\ \mathsf{0}\ X}\quad \frac{x : X \quad xs : \mathsf{Vec}\ n\ X}{x\ \overline{::}\ xs : \mathsf{Vec}\ (1{+}n)\ X}$$

$$\underline{\text{data}}\ \ \frac{n : \mathsf{Nat}}{\mathsf{Fin}\ n : \star}\ \ \underline{\text{where}}\ \ \frac{}{\overline{\mathsf{0}} : \mathsf{Fin}\ (1{+}n)}\quad \frac{i : \mathsf{Fin}\ n}{\overline{1+}\ i : \mathsf{Fin}\ (1{+}\ n)}$$

Dependently typed programming

$$\underline{\text{data}}\ \ \frac{n : \mathsf{Nat} \quad X : \star}{\mathsf{Vec}\ n\ X : \star}\ \ \underline{\text{where}}\ \ \frac{}{\overline{\mathsf{nil}} : \mathsf{Vec}\ 0\ X} \quad \frac{x : X \quad xs : \mathsf{Vec}\ n\ X}{x\ \overline{::}\ xs : \mathsf{Vec}\ (1{+}n)\ X}$$

$$\underline{\text{data}}\ \ \frac{n : \mathsf{Nat}}{\mathsf{Fin}\ n : \star}\ \ \underline{\text{where}}\ \ \frac{}{\overline{0} : \mathsf{Fin}\ (1{+}n)} \quad \frac{i : \mathsf{Fin}\ n}{\overline{1{+}}\ i : \mathsf{Fin}\ (1{+}\ n)}$$

$$\underline{\text{let}}\ \ \frac{xs : \mathsf{Vec}\ n\ X \quad i : \mathsf{Fin}\ n}{\mathbf{vnth}\ xs\ i : X}$$

Dependently typed programming

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat} \quad X : \star}{\mathsf{Vec}\ n\ X : \star}\ \underline{\text{where}}\ \frac{}{\overline{\mathsf{nil}} : \mathsf{Vec}\ 0\ X}\quad \frac{x : X \quad xs : \mathsf{Vec}\ n\ X}{x \mathbin{\overline{::}} xs : \mathsf{Vec}\ (1{+}n)\ X}$$

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat}}{\mathsf{Fin}\ n : \star}\ \underline{\text{where}}\ \frac{}{\overline{0} : \mathsf{Fin}\ (1{+}n)}\quad \frac{i : \mathsf{Fin}\ n}{\overline{1+}\ i : \mathsf{Fin}\ (1{+}\ n)}$$

$$\underline{\text{let}}\ \frac{xs : \mathsf{Vec}\ n\ X \quad i : \mathsf{Fin}\ n}{\mathbf{vnth}\ xs\ i : X}$$

$$\begin{array}{lllll}
\mathbf{vnth} & xs & i & \Leftarrow & \underline{\text{case}}\ i \\
\quad \mathbf{vnth} & xs & \overline{0} & \Leftarrow & \underline{\text{case}}\ xs \\
\quad\quad \mathbf{vnth} & x \mathbin{\overline{::}} xs & \overline{0} & \Rightarrow & x \\
\quad \mathbf{vnth} & xs & \overline{1{+}j} & \Leftarrow & \underline{\text{case}}\ xs \\
\quad\quad \mathbf{vnth} & x \mathbin{\overline{::}} xs & \overline{1{+}j} & \Rightarrow & \mathbf{vnth}\ xs\ j
\end{array}$$

Dependent datatypes

Dependent datatypes

Given $I : \mathbf{Set}$ we define the slice category $\mathbf{Set}/I$ as:

Objects $F : I \to \mathbf{Set}$

Morphisms $\mathbf{Set}/I(F, G) = \Pi i : I.(F\, i) \to (G\, i)$

Dependent datatypes

Given $I : \mathbf{Set}$ we define the slice category $\mathbf{Set}/I$ as:

Objects $F : I \to \mathbf{Set}$

Morphisms $\mathbf{Set}/I(F, G) = \Pi i : I.(F\,i) \to (G\,i)$

Dependent (inductive) datatypes arise as initial algebras of endofunctors on slice categories.

Dependent datatypes

Given I : **Set** we define the slice category **Set**$/I$ as:

Objects $F : I \to \mathbf{Set}$

Morphisms $\mathbf{Set}/I(F, G) = \Pi i : I.(F\,i) \to (G\,i)$

Dependent (inductive) datatypes arise as initial algebras of endofunctors on slice categories.
E.g. $\mathsf{Fin} = \mu F : \mathsf{Nat} \to \mathbf{Set}. T_{\mathsf{Fin}}\, F$, where

$$
\begin{aligned}
T_{\mathsf{Fin}} &: \mathbf{Set}/\mathsf{Nat} \to \mathbf{Set}/\mathsf{Nat} \\
T_{\mathsf{Fin}}\, F\, n &= \Sigma m : \mathsf{Nat}. n = 1 + m \\
&\quad + \Sigma m : \mathsf{Nat}.(n = 1 + m) \times (F\, m)
\end{aligned}
$$

Dependent Containers

Dependent Containers

Given I, J : **Set** a dependent container $S \triangleright P$: **Con** $I\ J$ is given by

- $S : J \to \mathbf{Set}$, a family of shapes,
- $P : \Pi j : J.(S\,j) \to I \to \mathbf{Set}$, an indexed family of positions.

Dependent Containers

Given $I, J : \mathbf{Set}$ a dependent container $S \triangleright P : \mathbf{Con}\, I\, J$ is given by

- $S : J \to \mathbf{Set}$, a family of shapes,
- $P : \Pi j : J.(S j) \to I \to \mathbf{Set}$, an indexed family of positions.

The extension of a dependent container is a functor on slices, that is $[\![S \triangleright P]\!] : \mathbf{Set}/I \to \mathbf{Set}/J$, on objects

$$[\![S \triangleright P]\!]\, F\, j = \Sigma s : S\, j.\Pi i : I.(P\, j\, s\, i) \to (F\, i).$$

Morphisms of dependent containers

Morphisms of dependent containers

Given two dependent containers $S \triangleright P, T \triangleright Q \ : \ \mathbf{Con}(I,\ J)$ a morphism $f \triangleright u$ is given by

- $f \ : \ \Pi j \ : \ J.(S\,j) \to T\,j$
- $u \in \Pi i \ : \ I.\Pi j \ : \ J.\Pi s \ : \ S\,j.Q\,j\,s\,i \to P\,j\,s\,i$

Morphisms of dependent containers

Given two dependent containers $S \triangleright P, T \triangleright Q \,:\, \mathbf{Con}(I,\, J)$ a morphism $f \triangleright u$ is given by

- $f \,:\, \Pi j \,:\, J.(S\, j) \to T\, j$
- $u \in \Pi i \,:\, I.\Pi j \,:\, J.\Pi s \,:\, S\, j.Q\, j\, s\, i \to P\, j\, s\, i$

The extension of a container morphism is a natural transformation which is given by the following family of maps (for $F \,:\, J \to \mathbf{Set}$):

$$\begin{aligned} [\![f \;\triangleright\; u]\!]\, F &: \Pi j \,:\, J.[\![S \;\triangleright\; P]\!]\, F\, j \to [\![T \;\triangleright\; Q]\!]\, F\, j \\ [\![f \;\triangleright\; u]\!]\, F\, j\, (s, h) &= (f\, j\, s, \lambda i.(h\, i) \circ (u\, i)) \end{aligned}$$

Representation theorem

Theorem : Every natural transformation (i.e. polymorphic function) between dependent containers can be represented as a dependent container morphisms.
$[\![\,]\!]$ is full and faithful.

Strictly positive dependent types?

Strictly positive dependent types?

- **Theorem: ?** All **strictly positive dependent types** are representable as dependent containers in any Martin-Löf category.

Strictly positive dependent types?

- **Theorem: ?** All **strictly positive dependent types** are representable as dependent containers in any Martin-Löf category.
- What is a dependent strictly positive type?

Strictly positive dependent types?

- **Theorem: ?** All **strictly positive dependent types** are representable as dependent containers in any Martin-Löf category.
- What is a dependent strictly positive type?
- **Inductive Schemes**, as in Luo's UTT or COQ's Type Theoy give rise to dependent containers.

Strictly positive dependent types?

- **Theorem: ?** All **strictly positive dependent types** are representable as dependent containers in any Martin-Löf category.
- What is a dependent strictly positive type?
- **Inductive Schemes**, as in Luo's UTT or COQ's Type Theoy give rise to dependent containers.
- Better: define a collection of combinators to generate **strictly positive dependent types**.

Application to schema checking

Application to schema checking

- Systems based on Type Theory like COQ, Agda, Epigram use schemes to characterize sound definitions of datatypes.

Application to schema checking

- Systems based on Type Theory like COQ, Agda, Epigram use schemes to characterize sound definitions of datatypes.
- Schema checking is complex, incomplete and potentially unsound.

Application to schema checking

- Systems based on Type Theory like COQ, Agda, Epigram use schemes to characterize sound definitions of datatypes.
- Schema checking is complex, incomplete and potentially unsound.
- Using dependent containers we can implement extensible schemes which produce evidence by translating the scheme into core Type Theory with W-types.

Application to schema checking

- Systems based on Type Theory like COQ, Agda, Epigram use schemes to characterize sound definitions of datatypes.
- Schema checking is complex, incomplete and potentially unsound.
- Using dependent containers we can implement extensible schemes which produce evidence by translating the scheme into core Type Theory with W-types.
- This requires a Type Theory with an extensional propositional equality (under development).

Dependent Signatures?

Our current approach doesn't capture inductive definitions like the definition of the syntax of Type Theory which simultanbously introduces:

$$
\begin{aligned}
\mathrm{Con} &: \mathbf{Set} \\
\mathrm{Ty} &: \mathrm{Con} \to \mathrm{Set} \\
\mathrm{Tm} &: \Pi\Gamma : \mathrm{Con}.(\mathrm{Ty}\,\Gamma) \to \mathbf{Set}
\end{aligned}
$$

Related work

Related work

- Dependent containers are closely related to polynomial functors, which have been investigated by Gambino and Hyland.

Related work

- Dependent containers are closely related to polynomial functors, which have been investigated by Gambino and Hyland.
- Initial algebras of unary dependent containers correspond to the Petersson and Synek's tree types.

Related work

- Dependent containers are closely related to polynomial functors, which have been investigated by Gambino and Hyland.
- Initial algebras of unary dependent containers correspond to the Petersson and Synek's tree types.
- The categoy of dependent containers is equivalent to the category of Interaction Structures investigated by Hancock,Hyvernat and Setzer.