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.

$$\{ \text{triangle}, \text{circle}, \text{square} \}$$
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. $X = \text{Nat}$ we can instantiate a container by
What to do with a container?

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

- Choosing a shape, e.g.
What to do with a container?

Given some payload \( X \), e.g. \( X = \text{Nat} \) 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 $\text{Set} \to \text{Set}$:

$$[[S \triangleright P]](X) = \sum_{s \in S} P(s) \to X$$
Extension of a container type

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

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

where

$$\Sigma a \in A.B(a) = \{(a, b) \mid a \in A \land b \in B(a)\}$$
Example of a container type
Example of a container type

data \( X : \ast \)  
List \( X : \ast \)  
where  
\( \text{nil} : \text{List} X \)  
\( x : X \)  
x = xs : List X
Example of a container type

\[
\begin{align*}
\text{data} & \quad \frac{X : \star}{\text{List}} \quad \frac{X : \star}{\text{List}} \\
\text{where} & \quad \frac{\text{nil} : \text{List} X}{\text{List}} \quad \frac{x : X}{\text{List} X} \quad \frac{xs : \text{List} X}{\text{List} X}
\end{align*}
\]

\[
\text{List} X = \mu Y.1 + X \times Y
\]
Example of a container type

\[
\text{data } \frac{X : \star}{\text{List } X : \star} \quad \text{where} \quad \frac{\text{nil} : \text{List } X}{x : X \quad xs : \text{List } X} \quad \text{List } X = \mu Y. 1 + X \times Y
\]

\[
\text{List } X \simeq \Sigma n \in \text{Nat.}\{i < n\} \to X
\]
Example of a container type

\[
\begin{align*}
\text{data} & \quad \frac{X : \star}{\text{List } X} : \star \quad \text{where} \quad \frac{\text{nil} : \text{List } X \quad x : X \quad xs : \text{List } X}{x = xs : \text{List } X} \\
\text{List } X & = \mu Y.1 + X \times Y \\
\text{List } X & \simeq \Sigma n \in \text{Nat.} \{ i < n \} \rightarrow X \\
& = \Sigma n \in \text{Nat.} n \rightarrow X
\end{align*}
\]
$n$-ary containers
$n$-ary containers

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

- $S : \text{Set}$
- $\vec{P} : n \rightarrow S \rightarrow \text{Set}$
\textit{n}-ary containers

An \textit{n}-ary container \( S \triangleright \vec{P} \) is given by

- \( S : \text{Set} \)
- \( \vec{P} : n \rightarrow S \rightarrow \text{Set} \)

Its extension is an endofunctor \( \text{Set}^n \rightarrow \text{Set} \) is:

\[
[S \triangleright \vec{P}](X) = \sum_{s \in S} \prod_{i < n} P_{i \, s} \rightarrow X \, i
\]
Morphisms of containers
Morphisms of containers

Given containers

\[ F(X) = \Sigma s \in S.P(s) \to X \]
\[ G(X) = \Sigma t \in T.Q(t) \to X \]

a morphism \( f \circ u \in \text{Con}(F, G) \) is given by
Morphisms of containers

Given containers

\[ F(X) = \Sigma s \in S.P(s) \to X \]
\[ G(X) = \Sigma t \in T.Q(t) \to X \]

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

\[ f \in S \to T \]
Morphisms of containers

Given containers

\[ F(X) = \Sigma s \in S.P(s) \rightarrow X \]
\[ G(X) = \Sigma t \in T.Q(t) \rightarrow X \]

a morphism \( f \circ u \in \textbf{Con}(F, G) \) is given by

\[ f \in S \rightarrow T \]
\[ u \in \Pi s \in S.Q(f(s)) \rightarrow P(s) \]
Morphisms of containers

Given containers

\[ F(X) = \Sigma s \in S.P(s) \to X \]
\[ G(X) = \Sigma t \in T.Q(t) \to X \]

A morphism \( f \triangleright u \in \text{Con}(F, G) \) is given by

\[ f \in S \to T \]
\[ u \in \Pi s \in S.Q(f(s)) \to P(s) \]
\[ [f \triangleright u]_X \in F(X) \to G(X) \]
Morphisms of containers

Given containers

\[ F(X) = \Sigma s \in S.P(s) \rightarrow X \]
\[ G(X) = \Sigma t \in T.Q(t) \rightarrow X \]

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

\[ f \in S \rightarrow T \]
\[ u \in \Pi s \in S.Q(f(s)) \rightarrow P(s) \]
\[ \llbracket f \triangleright u \rrbracket_X \in F(X) \rightarrow G(X) \]
\[ = (s, h) \mapsto (f(s), h \circ u(s)) \]
Representation theorem

Theorem: Every natural transformation (i.e. polymorphic function) between containers can be represented as a container morphism.

Example: any natural transformation is given
Theorem: Every natural transformation (i.e. polymorphic function) between containers can be represented as a container morphisms. \( \square \) is full and faithful.
**Theorem**: Every natural transformation (i.e. polymorphic function) between containers can be represented as a container morphism. $\mathcal{C}$ is full and faithful.

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

\[
\begin{align*}
  f &: \text{Nat} \to \text{Nat} \\
  u &: \prod n : \text{Nat}. (f n) \to n
\end{align*}
\]
Strictly positive types

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.
Strictly positive types

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

- **Strictly positive types** are generated by $0, 1, +, \times, C \to -$ (constant exponentiation), $\mu$ (initial algebra) and $\nu$ (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 $0, 1, +, \times, C \to -$ (constant exponentiation), $\mu$ (initial algebra) and $\nu$ (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 $0, 1, +, \times, C \rightarrow -$ (constant exponentation), $\mu$ (initial algebra) and $\nu$ (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

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.
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

data $n : \text{Nat} \quad X : \star$  where  

\[
\begin{align*}
\text{nil} & : \text{Vec } 0 \quad X \\
x \equiv xsl & : \text{Vec } (1+n) \quad X
\end{align*}
\]
Dependently typed programming

```
data  \frac{n : \text{Nat}}{\text{Vec} \ n \ X : *} \quad \text{where} \quad \frac{x : X \ \ xs : \text{Vec} \ n \ X}{\text{nil} : \text{Vec} \ 0 \ X}
```

```
data  \frac{n : \text{Nat}}{\text{Fin} \ n : *} \quad \text{where} \quad \frac{0 : \text{Fin} \ (1+n)}{1+i : \text{Fin} \ (1+n)}
```
Dependently typed programming

\[
\begin{align*}
\text{data} \quad & n : \text{Nat} \quad X : \star \quad \text{where} \quad \frac{}{\text{Vec} \; n \; X : \star} \quad \frac{}{\text{nil} : \text{Vec} \; 0 \; X} \quad \frac{x : X \quad xs : \text{Vec} \; n \; X}{x \mapsto xs : \text{Vec} \; (1+n) \; X} \\
\text{data} \quad & n : \text{Nat} \quad \frac{n : \text{Nat}}{\text{Fin} \; n : \star} \quad \text{where} \quad \frac{}{\text{Fin} \; (1+n) : \star} \quad \frac{}{0 : \text{Fin} \; (1+n)} \quad \frac{i : \text{Fin} \; n}{1+i : \text{Fin} \; (1+n)} \\
\text{let} \quad & xs : \text{Vec} \; n \; X \quad i : \text{Fin} \; n \quad \text{where} \quad \frac{}{\text{nth} \; xs \; i : X}
\end{align*}
\]
Dependently typed programming

data \( n : \text{Nat} \) \( X : \star \) where
\[ \begin{array}{rcl}
\text{Vec} n X & : & \star \\
\text{nil} & : & \text{Vec} 0 X \\
\text{x \in xs} & : & \text{Vec} (1 + n) X
\end{array} \]

\[
data \quad n : \text{Nat} \quad \text{Fin} n : \star \quad \text{where}
\begin{array}{rcl}
\text{0} & : & \text{Fin} (1 + n) \\
\text{1 + i} & : & \text{Fin} (1 + n)
\end{array}
\]

let \( xs : \text{Vec} n X \) \( i : \text{Fin} n \) \( \text{vnth} \) \( xs \) \( i \) \( \Leftarrow \) case \( i \)
\[ \begin{array}{rcl}
\text{vnth} & \quad x & \quad 0 & \Leftarrow \text{case} \quad xs \\
\text{vnth} & \quad x & \quad \overline{0} & \Rightarrow \quad x \\
\text{vnth} & \quad x & \quad \overline{1 + j} & \Leftarrow \text{case} \quad xs \\
\text{vnth} & \quad x & \quad \overline{1 + j} & \Rightarrow \quad \text{vnth} \quad xs \quad j
\end{array} \]
Dependent datatypes
Dependent datatypes

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

**Objects** $F : I \to \text{Set}$

**Morphisms** $\text{Set}/I(F, G) = \prod i : I.(F i) \to (G i)$
Dependent datatypes

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

**Objects** $F : I \to \text{Set}$

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

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

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

**Objects** $F : I \to \text{Set}$

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

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

E.g. $\text{Fin} = \mu F : \text{Nat} \to \text{Set}. T_{\text{Fin}} F$, where

$$T_{\text{Fin}} : \text{Set}/\text{Nat} \to \text{Set}/\text{Nat}$$

$$T_{\text{Fin}} F n = \Sigma m : \text{Nat}. n = 1 + m$$

$$+ \Sigma m : \text{Nat}. (n = 1 + m) \times (F m)$$
Dependent Containers

Given a dependent container is given by \( \mathcal{C} \), a family of shapes, \( \mathcal{F} \), an indexed family of positions. The extension of a dependent container is a functor on slices, that is, on objects.
Dependent Containers

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

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

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

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

The extension of a dependent container is a functor on slices, that is $\llbracket S \triangleright P \rrbracket : \text{Set} / I \rightarrow \text{Set} / J$, on objects

$$\llbracket S \triangleright P \rrbracket F j = \Sigma s : S j. \Pi i : I. (P j s i) \rightarrow (F i).$$
Morphisms of dependent containers
Morphisms of dependent containers

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

- $f : \Pi j : J.(S j) \rightarrow T j$
- $u \in \Pi i : I.\Pi j : J.\Pi s : S j.Q j s i \rightarrow P j s i$
Morphisms of dependent containers

Given two dependent containers $S \triangleright P, T \triangleright Q : \text{Con}(I, J)$ 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 $F : J \to \text{Set}$):

$$
[f \triangleright u] F : \Pi j : J.\lbrack S \triangleright P \rbrack F j \to \lbrack T \triangleright Q \rbrack F j
$$
$$
[f \triangleright u] F j (s, h) = (f j s, \lambda i.(h i) \circ (u i))
$$
**Theorem** : Every natural transformation (i.e. polymorphic function) between dependent containers can be represented as a dependent container morphisms. \[\mathcal{F}\] is full and faithful.
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 Theory give rise to dependent containers. Better: define a collection of combinators to generate 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 Theory 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 Theory give rise to dependent containers.
- Better: define a collection of combinators to generate strictly positive dependent 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).
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 simultaneously introduces:

\[
\begin{align*}
\text{Con} &: \text{Set} \\
\text{Ty} &: \text{Con} \rightarrow \text{Set} \\
\text{Tm} &: \Pi \Gamma : \text{Con.}(\text{Ty } \Gamma) \rightarrow \text{Set}
\end{align*}
\]
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 category of dependent containers is equivalent to the category of Interaction Structures investigated by Hancock, Hyvernat and Setzer.
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 category of dependent containers is equivalent to the category of Interaction Structures investigated by Hancock, Hyvernat and Setzer.