Towards a theory of higher inductive types

Thorsten Altenkirch\textsuperscript{1} Paulo Capriotti\textsuperscript{1} \textit{Gabe Dijkstra}\textsuperscript{1}  
Fredrik Nordvall Forsberg\textsuperscript{2}

\textsuperscript{1}University of Nottingham
\textsuperscript{2}University of Strathclyde

May 20th, 2015
Goal

Our goal is to:

- Define a general class of higher inductive types
- Akin to W-types
- Building upon Shulman and Lumsdaine’s semantics
Ordinary inductive types:

```
data T : Type where
    c_0 : F_0 T → T
    ...
    c_k : F_k T → T
```

where every $F_i : Type → Type$ is a (strictly positive) functor.
Higher inductive types versus ordinary inductive types

Ordinary inductive types:

\[
\text{data } T : \text{Type where} \\
c : F_0\ T + \ldots + F_k\ T \to T
\]

where every \( F_i : \text{Type} \to \text{Type} \) is a (strictly positive) functor.
Higher inductive types versus ordinary inductive types

Ordinary inductive types:

```
data T : Type where
  c : F T → T
```

where $F : Type \rightarrow Type$ is a (strictly positive) functor.
Higher inductive types versus ordinary inductive types

Higher inductive types, e.g. the circle:

```haskell
data S¹ : Type where
  base : S¹
  loop : base =_{S¹} base
```

- Dependencies on previous constructors
- *Higher* constructors: target of constructors not always $T$, but can also be an iterated path space of $T$.

Single functor $Type \to Type$ no longer suffices
Higher inductive types versus ordinary inductive types

Higher inductive types, e.g. propositional truncation:

```
data ||A|| : Type where
  [_] : A → ||A||
  trunc : (x y : ||A||) → x = y
```

- Dependencies on previous constructors
- *Higher* constructors: target of constructors not always $T$, but can also be an iterated path space of $T$.

Single functor $Type → Type$ no longer suffices
General framework

Constructors are dependent dialgebras:

\[
\text{data } T : \text{Type where}
\]
\[
c_0 : (x : F_0 \ T) \rightarrow G_0 (T, x)
\]
\[
c_1 : (x : F_1 (T, c_0)) \rightarrow G_1 ((T, c_0), x)
\]
\[
\vdots
\]
\[
c_k : (x : F_k (T, c_0, \ldots, c_{k-1})) \rightarrow G_k ((T, c_0, \ldots, c_{k-1}), x)
\]

We will call:

- Every \( F_i \) an argument functor
- Every \( G_i \) a target functor
General framework – example: *interval*

The interval type:

```plaintext
data I : Type where
  zero : I
  one : I
  seg : zero = one
```

Argument functors:

```plaintext
F_0 X :≡ 1 \quad (F_0 : Type \rightarrow Type)
F_1 (X, z) :≡ 1 \quad (F_1 : (F_0, G_0)\text{-alg} \rightarrow Type)
F_2 (X, z, o) :≡ 1 \quad (F_2 : (F_1, G_1)\text{-alg} \rightarrow Type)
```

Target functors:

```plaintext
G_0 (X, x) :≡ X \quad (G_0 : \int_{\text{Type}} F_0 \rightarrow Type)
G_1 ((X, z), x) :≡ X \quad (G_1 : \int_{(F_0, G_1)\text{-alg}} F_1 \rightarrow Type)
G_2 ((X, z, o), x) :≡ (z = o) \quad (G_2 : \int_{(F_1, G_1)\text{-alg}} F_2 \rightarrow Type)
```
General framework – example: \textit{interval}

The interval type:

\begin{verbatim}
data l : Type where
  zero : l
  one : l
  seg : zero = one
\end{verbatim}

\textit{Category of elements:}

\textbf{objects:}
\((X : Type) \times F_0 X\)

\textbf{morphisms} \((X, x) \rightarrow (Y, y)\):
\((f : X \rightarrow Y) \times (F_0 f x = y)\)

Argument functors:
\begin{align*}
  F_0 & \equiv 1 \quad (F_0 : Type \rightarrow Type) \\
  F_1 & \equiv 1 \quad (F_1 : (F_0, G_0)\text{-alg} \rightarrow Type) \\
  F_2 & \equiv 1 \quad (F_2 : (F_1, G_1)\text{-alg} \rightarrow Type)
\end{align*}

Target functors:
\begin{align*}
  G_0 & \equiv X \quad (G_0 : \int_{Type} F_0 \rightarrow Type) \\
  G_1 & \equiv X \quad (G_1 : \int_{(F_0, G_1)\text{-alg}} F_1 \rightarrow Type) \\
  G_2 & \equiv (z = o) \quad (G_2 : \int_{(F_1, G_1)\text{-alg}} F_2 \rightarrow Type)
\end{align*}
Constructors are \textit{dependent dialgebras}:

\[ c : (x : F X) \rightarrow G(X, x) \]

where

- \( \mathbb{C} : \text{Cat} \)
- \( F : \mathbb{C} \rightarrow \text{Type} \) (\textit{argument} functor)
- \( G : \int_{\mathbb{C}} F \rightarrow \text{Type} \) (\textit{target} functor)
General framework – 0-constructors

0-constructors are of the form:

\[ c : (x : F X) \to U X \]

where

- \( C : \text{Cat} \) with a forgetful functor \( U : C \to \text{Type} \)
- \( F : C \to \text{Type} \)
- \( G : \int_C F \to \text{Type} \)

\[ G (X, x) \equiv U X \]
General framework – 1-constructors

1-constructors are of the form:

\[ c : (x : F X) \rightarrow (l_0 X x = r_0 X x) \]

where

- \( C : \text{Cat} \) with a forgetful functor \( U : C \rightarrow \text{Type} \)
- \( F : C \rightarrow \text{Type} \)
- \( G : \int C F \rightarrow \text{Type} \)
- \( l_0, r_0 : F \rightarrow U \)

\[ G (X, x) \equiv (l_0 X x = r_0 X x) \]

We call this \( G \) functor \( Eq_0 \)
General framework – 2-constructors

For 2-constructors:

\[ c : (x : F X) \rightarrow (l_1 X x = r_1 X x) \]

where

- \( l_0 r_0 : F \rightarrow U \)  
  (with \( Eq_0 (X, x) \equiv (l_0 X x = r_0 X x) \))
- \( l_1 r_1 : 1 \rightarrow Eq_0 \)  
  (with \( Eq_1 (X, x) \equiv (l_1 X x = r_1 X x) \))
General framework – 3-constructors

For 3-constructors:

\[ c : (x : F \ X) \rightarrow (l_2 \ X \ x = r_2 \ X \ x) \]

where

- \( l_0 \ r_0 : F \rightarrow U \) (with \( Eq_0 \ (X, x) :\equiv (l_0 \ X \ x = r_0 \ X \ x) \))
- \( l_1 \ r_1 : 1 \rightarrow Eq_0 \) (with \( Eq_1 \ (X, x) :\equiv (l_1 \ X \ x = r_1 \ X \ x) \))
- \( l_2 \ r_2 : 1 \rightarrow Eq_1 \) (with \( Eq_2 \ (X, x) :\equiv (l_2 \ X \ x = r_2 \ X \ x) \))
General framework – \((n + 1)\)-constructors

For \((n + 1)\)-constructors:

\[ c : (x : F X) \rightarrow (l_n X x = r_n X x) \]

where

- \( l_0 \ r_0 : F \rightarrow U \) (with \( Eq_0 (X, x) \equiv (l_0 X x = r_0 X x) \))
- \( l_1 \ r_1 : 1 \rightarrow Eq_0 \) (with \( Eq_1 (X, x) \equiv (l_1 X x = r_1 X x) \))
- \( l_2 \ r_2 : 1 \rightarrow Eq_1 \) (with \( Eq_2 (X, x) \equiv (l_2 X x = r_2 X x) \))
- \( \vdots \)
- \( l_n \ r_n : 1 \rightarrow Eq_{n-1} \) (with \( Eq_n (X, x) \equiv (l_n X x = r_n X x) \))
Strict positivity – ordinary inductive types

We can’t allow any argument functor: it has to be strictly positive:

\begin{verbatim}
data Term : Type where
   lam : (Term → Term) → Term
\end{verbatim}
Strict positivity – higher inductive types

We can’t allow any argument functor: it has to be strictly positive:

```
data InitialField : Type where
  0 : InitialField
  1 : InitialField
  _+_ : InitialField → InitialField → InitialField
  _*_ : InitialField → InitialField → InitialField
  ...
  _⁻¹ : (x : InitialField) → (x = 0 → ⊥) → InitialField
  ...
```

`InitialField` does not exist: `_⁻¹` is not strictly positive
**Type-containers**

Strictly positive functors $\text{Type} \rightarrow \text{Type}$: containers

- **Shapes $S : \text{Type}$**
- **Positions $T : S \rightarrow \text{Type}$**

\[
[S \triangleleft P]_0 : \mathbb{C} \rightarrow \text{Type}
\]
\[
[S \triangleleft P]_0 X : \equiv (s : S) \times (P s \rightarrow X)
\]

\[
[S \triangleleft P]_1 : (X \rightarrow Y) \rightarrow [S \triangleleft P]_0 X \rightarrow [S \triangleleft P]_0 Y
\]
\[
[S \triangleleft P]_1 f (s, t) : \equiv (s, f \circ t)
\]

**Example:**

\[
\text{const}_A X = [A \triangleleft (\lambda a.0)]_0 X
\]
\[
= A \times (0 \rightarrow X)
\]
\[
= A
\]
\(\text{\text{C}}\)-containers

Strictly positive functors \(\text{\text{C}} \to \text{Type}\): \(\text{\text{C}}\)-containers (or familially representable)

- Shapes \(S : \text{Type}\)
- Positions \(T : S \to |\text{\text{C}}|\)

\[
\begin{align*}
\llbracket S \triangleleft P \rrbracket_0 : \text{\text{C}} \to \text{Type} \\
\llbracket S \triangleleft P \rrbracket_0 X & \equiv (s : S) \times \text{\text{C}} (P s, X) \\
\llbracket S \triangleleft P \rrbracket_1 : \text{\text{C}} (X, Y) \to \llbracket S \triangleleft P \rrbracket_0 X \to \llbracket S \triangleleft P \rrbracket_0 Y \\
\llbracket S \triangleleft P \rrbracket_1 f (s, t) & \equiv (s, f \circ t)
\end{align*}
\]

Example (assuming \(0 : |\text{\text{C}}|\) is initial):

\[
\text{\text{const}}_A X = \llbracket A \triangleleft (\lambda a.0) \rrbracket_0 X \\
= A \times \text{\text{C}} (0, X) \\
= A
\]
\(\mathcal{C}\)-container morphisms

- Data for higher constructors requires *natural transformations*
- Natural transformations between containers: *container morphisms*:

For containers \( S \triangleleft P \) and \( T \triangleleft Q \), container morphisms are:

\[
(f : S \to T) \times (g : (a : S) \to \mathbb{C}(Q(fa), Pa))
\]

Container morphisms are *complete*:

- Each container morphism gives rise to a natural transformation and vice versa
Expressivity of containers

Data for constructors can be given using containers and container morphisms:

- Argument functors are given as containers
- Forgetful functors $U_i : (F_i, G_i)\text{-alg} \to \text{Type}$ can be given as containers if there exist $L_i \dashv U_i$
- Data for $Eq_n$ functors are given as container morphisms
- $Eq_n$ functors can be given as containers if we have $(n + 1)$-HITs
Simplified approach to 1-HITs

In practice, constructor arguments rarely seem to refer to previous constructors.

We can identify a class of 1-HITs where we have:

- 0-constructors which do not depend on other constructors
- 1-constructors which may only depend on 0-constructors in the targets
- No dependencies between the 1-constructors

Examples: circle, suspension, truncation

Non-example: hub-spokes version of the torus
Simplified approach to 1-HITs

If we have:
- $F_0 : \text{Type} \rightarrow \text{Type}$ with $U_0 : (F_0, G_0)\text{-alg} \rightarrow \text{Type}$
- $F_1 : (F_0, G_0)\text{-alg} \rightarrow \text{Type}$ such that $F_1 = F'_1 \circ U_0$
- ... where $F'_1 : \text{Type} \rightarrow \text{Type}$

then

$$F_1 \rightarrow U_0 \simeq F'_1 \rightarrow F_0^*$$

where $F_0^*$ is the free monad of $F_0$.

This approach has been fully formalised in Agda.
Coherence

- We can use \( \mathbb{C} \)-containers to internalise the theory
- We still need to be able to talk about the categories of dependent dialgebras
Coherence – Category of dependent dialgebras

Given $\mathcal{C}$ a category with functors $F : \mathcal{C} \to Type$ and $G : \int_{\mathcal{C}} F \to Type$, we can consider the category of dialgebras $(F, G)$-alg:

- objects:
  
  $$(X : |\mathcal{C}|) \times (\theta : (x : F X) \to G (X, x))$$

- morphisms $(X, \theta) \to (Y, \rho)$:

  $$(f : \mathcal{C} (X, Y)) \times \text{comm} : (x : F X) \to G (f, \text{refl}) (\theta x) = \rho (F f x)$$
Coherence

- We can use \( \mathbb{C} \)-containers to internalise the theory
- We still need to be able to talk about the categories of dependent dialgebras

Design choices:

- Define the categories with strict equality and possibly lose some expressivity (e.g. the torus)
- Work with appropriately defined \((\infty, 1)\)-categories and deal with the coherence issues, that increase with the number of constructors
Conclusions

- Higher inductive types are sequences of dependent dialgebras
- $\mathbb{C}$-containers allow us to formalise the data needed to define constructors
- A simplified approach to 1-HITs has been successfully formalised
- Coherence problems increase with the number of constructors
- We can work in a type theory with strict equality and avoid coherence problems but we lose some expressivity