Codata

Thorsten Altenkirch
University of Nottingham
Haskell: data = codata?
Haskell: data = codata ?

\[ \text{data List} = \text{Nil} \mid \text{Cons Nat List} \]
Haskell: data = codata ?

\[
\text{data List} = \text{Nil} \mid \text{Cons Nat List}
\]

\[
even \in \text{List} \rightarrow \text{Bool}
\]

\[
even \text{ Nil} = \text{True}
\]

\[
even (\text{Cons } a \ as) = \neg (even as)
\]
Haskell: data = codata?

\[
\text{data List} = \text{Nil} \mid \text{Cons Nat List}
\]

\[
even \in \text{List} \rightarrow \text{Bool}
\]
\[
even \text{ Nil} = \text{True}
\]
\[
even (\text{Cons } a \text{ as}) = \neg (even \text{ as})
\]

\[
\text{from} \in \text{Nat} \rightarrow \text{List}
\]
\[
\text{from } n = \text{Cons } n \ (\text{from } (n + 1))
\]
Haskell: data = codata ?

\[
data \text{ List} = \text{Nil} \mid \text{Cons Nat List}
\]

\[
even \in \text{List} \rightarrow \text{Bool}
\]

\[
even \text{ Nil} = \text{True}
\]

\[
even (\text{Cons } a \ as) = \neg (\text{even } as)
\]

\[
\text{from} \in \text{Nat} \rightarrow \text{List}
\]

\[
\text{from } n = \text{Cons } n (\text{from } (n + 1))
\]

\[
even (\text{from } 0) \text{ diverges!}
\]
Type Theory: data ≠ codata
Type Theory: data ≠ codata

data List = Nil | Cons Nat List

codata List\(\infty\) = Nil\(\infty\) | Cons\(\infty\) Nat List\(\infty\)
Type Theory: data $\neq$ codata

\[
\text{data List} = \text{Nil} \mid \text{Cons Nat List}
\]
\[
\text{codata List}^\infty = \text{Nil}^\infty \mid \text{Cons}^\infty \text{Nat List}^\infty
\]

\[
even \in \text{List} \rightarrow \text{Bool}
\]
\[
even \text{Nil} = \text{True}
\]
\[
even (\text{Cons } a \ as) = \neg (even \ as)
\]
Type Theory: data ≠ codata

data List = Nil | Cons Nat List

codata List = Nil | Cons Nat List

even ∈ List → Bool

even Nil = True

even (Cons a as) = ¬ (even as)

from ∈ Nat → List

from n = Cons n (from (n + 1))
Type Theory: data ≠ codata

data List = Nil | Cons Nat List

codata List^\infty = Nil^\infty | Cons^\infty Nat List^\infty

even ∈ List → Bool

even Nil = True
even (Cons a as) = ¬ (even as)

from ∈ Nat → List^\infty

from n = Cons^\infty n (from (n + 1))
even (from 0) doesn’t typecheck.
codata in Type Theory
codata in Type Theory

Thierry Coquand

*Infinite Objects in Type Theory*

TYPES 93
codata in Type Theory

Thierry Coquand
_Infinite Objects in Type Theory_
TYPES 93

Eduardo Giminez
_Coinductive Types in COQ_
93 – 95
see Coq’Art, pp.347 – 376
Codata?

Codata seems more exotic than data. Categorically codata (terminal coalgebras) is a dual of data (initial algebras). Proposal: a conceptual duality based on contracts which justifies Observational Type Theory reflecting this symmetry.
Codata seems more exotic than data.
Codata?

- Codata seems more exotic than data.
- Categorically codata (terminal coalgebras) is a dual of data (initial algebras).
Codata?

- Codata seems more exotic than data.
- Categorically codata (terminal coalgebras) is a dual of data (initial algebras)
- Proposal: a conceptual duality based on contracts
Codata?

- Codata seems more exotic than data.
- Categorically codata (terminal coalgebras) is a dual of data (initial algebras).
- Proposal: a conceptual duality based on contracts.
- which justifies *Observational Type Theory* reflecting this symmetry.
Data – the producer contract

The producer of data promises that he/she will construct data only using the agreed constructors.

Consequences:
- pattern matching
- structural recursion
- induction as structural recursion on proofs
Data – the producer contract

The producer of data promises that he/she will construct data only using the agreed constructors.
Data – the producer contract

The producer of data promises that he/she will construct data only using the agreed constructors.

Consequences:
Data – the producer contract

The producer of **data** promises that he/she will construct data only using the agreed constructors.

**Consequences:**
- pattern matching
Data – the producer contract

The producer of data promises that he/she will construct data only using the agreed constructors.

Consequences:

- pattern matching
- structural recursion
Data – the producer contract

The producer of **data** promises that he/she will construct data only using the agreed constructors.

**Consequences:**

- pattern matching
- structural recursion
- induction as structural recursion on proofs
Codata – the consumer contract

The consumer of codata promises that he/she will only analyze codata using the patterns induced by the agreed constructors.

Consequences:
- constructors
- guarded corecursion
- coinduction as guarded recursion on proofs
The consumer of codata promises that he/she will only analyze codata using the patterns induced by the agreed constructors.
The consumer of **codata** promises that he/she will only analyze codata using the patterns induced by the agreed constructors.

**Consequences:**
The consumer of codata promises that he/she will only analyze codata using the patterns induced by the agreed constructors.

Consequences:
- constructors
Codata – the consumer contract

The consumer of codata promises that he/she will only analyze codata using the patterns induced by the agreed constructors.

Consequences:
- constructors
- guarded corecursion
The consumer of codata promises that he/she will only analyze codata using the patterns induced by the agreed constructors.

Consequences:
- constructors
- guarded corecursion
- coinduction as guarded recursion on proofs
A simple proposition
A simple proposition

\[
\begin{align*}
mapS & \in \text{List}^\infty \rightarrow \text{List}^\infty \\
mapS \Nil^\infty & = \Nil^\infty \\
mapS \text{Cons}^\infty n \tilde{n} & = \text{Cons}^\infty (n + 1) (\mapS \tilde{n})
\end{align*}
\]
A simple proposition

\[ \text{let } n \in \text{Nat} \implies \text{lem } n \in \text{mapS (from } n) = \text{from } (n + 1) \]

\[ \text{mapS } \in \text{List}^\infty \to \text{List}^\infty \]

\[ \text{mapS } \text{Nil}^\infty = \text{Nil}^\infty \]

\[ \text{mapS } \text{Cons}^\infty n \bar{n} = \text{Cons}^\infty (n + 1) (\text{mapS } \bar{n}) \]
A simple proposition

\[\text{mapS} \in \text{List}^\infty \rightarrow \text{List}^\infty\]

\[\text{mapS}\ Nil^\infty = \text{Nil}^\infty\]

\[\text{mapS}\ \text{Cons}^\infty\ n\ \tilde{n} = \text{Cons}^\infty\ (n + 1)\ (\text{mapS}\ \tilde{n})\]

\[
\text{let}\quad n \in \text{Nat}\\
\text{lem}\ n \in \text{mapS}\ (\text{from}\ n) = \text{from}\ (n + 1)
\]

* Let’s have a closer look at \(=\).*
Equality for List
Equality for List

\[
\text{data } \frac{\vec{m}, \vec{n} \in \text{List}}{\vec{m} = \vec{n} \in \text{Prop}} \quad \text{where}
\]
Equality for List

\[
\text{data} \quad \frac{\bar{m}, \bar{n} \in \text{List}}{\bar{m} = \bar{n} \in \text{Prop}} \quad \text{where} \quad \text{EqNil} \in \text{Nil} = \text{Nil}
\]
Equality for List

\[
\text{data} \quad \frac{\vec{m}, \vec{n} \in \text{List}}{\vec{m} = \vec{n} \in \text{Prop}} \quad \text{where}
\]

\[
\text{EqNil} \in \text{Nil} = \text{Nil}
\]

\[
\frac{p \in m = n \quad \vec{p} \in \vec{m} = \vec{n}}{\text{EqCons} \quad p \vec{p} \in \text{Cons} \quad m \vec{m} = \text{Cons} \quad n \vec{n}}
\]
Properties of
Properties of $=$

\[
\begin{align*}
\text{let} & \quad \vec{n} \in \text{List} \\
\text{refl} \vec{n} & \in \vec{n} = \vec{n}
\end{align*}
\]
Properties of $=\!=$

Let

\[
\bar{n} \in \text{List} \\
\text{refl } \bar{n} \in \bar{n} = \bar{n}
\]

\[
\text{refl } \text{Nil} = \text{EqNil}
\]

\[
\text{refl } (\text{Cons } n \bar{n}) = \text{EqCons } (\text{refl } n) (\text{refl } \bar{n})
\]
Properties of \(\equiv\)

\[
\begin{align*}
\text{let } & n \in \text{List} \\
\frac{\text{refl } n \in \langle n \rangle = n}{\text{refl } n \in \langle n \rangle = n}
\end{align*}
\]

\[
\begin{align*}
\text{let } & n \in \text{List} \\
\frac{\text{refl } n \in \langle n \rangle = n}{\text{refl } n \in \langle n \rangle = n}
\end{align*}
\]

\[
\begin{align*}
\text{refl } \text{Nil} & = \text{EqNil} \\
\text{refl } (\text{Cons } n \langle n \rangle) & = \text{EqCons } (\text{refl } n) (\text{refl } \langle n \rangle)
\end{align*}
\]

\[
\begin{align*}
\text{let } & \rho \in \langle m \rangle = \langle n \rangle \quad \sigma \in \langle n \rangle = \langle o \rangle \\
\frac{\text{trans } \rho \sigma \in \langle m \rangle = \langle o \rangle}{\text{trans } \rho \sigma \in \langle m \rangle = \langle o \rangle}
\end{align*}
\]
Properties of $\equiv$

Let $\vec{n} \in \text{List}$

\[
\begin{align*}
\text{refl } \vec{n} & \in \vec{n} = \vec{n} \\
\text{refl } \text{Nil} & = \text{EqNil} \\
\text{refl } (\text{Cons } n \, \vec{n}) & = \text{EqCons } (\text{refl } n) \, (\text{refl } \vec{n})
\end{align*}
\]

Let $\vec{p} \in \vec{m} = \vec{n} \quad \vec{q} \in \vec{n} = \vec{o}$

\[
\begin{align*}
\text{trans } \vec{p} \, \vec{q} & \in \vec{m} = \vec{o} \\
\text{trans } \text{EqNil} & = \text{EqNil} \\
\text{trans } (\text{EqCons } p \, \vec{p}) \, (\text{EqCons } q \, \vec{p}) & = \text{EqCons } (\text{trans } p \, q) \, (\text{trans } \vec{p} \, \vec{q})
\end{align*}
\]
Equality for List
Equality for List\(^\infty\)

\[
\text{codata} \quad \frac{\vec{m}, \vec{n} \in \text{List}^\infty}{\vec{m} = \vec{n} \in \text{Prop}} \quad \text{where}
\]
Equality for List$^\infty$

**Codata**

\[
\begin{align*}
\text{codata} & \quad \tilde{m}, \tilde{n} \in \text{List}^\infty \\
\text{} & \quad \tilde{m} = \tilde{n} \in \text{Prop} \\
\text{EqNil}^\infty & \in \text{Nil}^\infty = \text{Nil}^\infty
\end{align*}
\]
Equality for List\(^\infty\)

\[
\text{codata } \quad \frac{\vec{m}, \vec{n} \in \text{List}^\infty}{\vec{m} = \vec{n} \in \text{Prop}} \quad \text{where}
\]

\[
\text{EqNil}^\infty \in \text{Nil}^\infty = \text{Nil}^\infty
\]

\[
\frac{p \in m = n \quad \vec{p} \in \vec{m} = \vec{n}}{\text{EqCons}^\infty \quad p \vec{p} \in \text{Cons}^\infty \quad m \vec{m} = \text{Cons}^\infty \quad n \vec{n}}
\]
Properties of
Properties of $=$

\[
\text{let } \overline{\bar{n}} \in \text{List}^\infty \\
\text{refl } \overline{\bar{n}} \in \overline{\bar{n}} = \overline{\bar{n}}
\]
Properties of $\mathrel{=}$

\[
\begin{align*}
\tilde{n} & \in \textbf{List}^\infty \\
\text{let} \quad \text{refl } \tilde{n} & \in \tilde{n} = \tilde{n} \\
\text{refl } \text{Nil}^\infty & = \text{EqNil}^\infty \\
\text{refl } (\text{Cons}^\infty \ n \ \tilde{n}) & = \text{EqCons}^\infty (\text{refl } n) (\text{refl } \tilde{n})
\end{align*}
\]
Properties of \( = \)

\[
\begin{align*}
\text{let} & \quad \tilde{n} \in \text{List}^\infty \\
& \quad \text{refl } \tilde{n} \in \tilde{n} = \tilde{n} \\
& \quad \text{refl } \text{Nil}^\infty = \text{EqNil}^\infty \\
& \quad \text{refl } (\text{Cons}^\infty \ n \ \tilde{n}) = \text{EqCons}^\infty (\text{refl } \ n) (\text{refl } \tilde{n}) \\
\text{let} & \quad \tilde{p} \in \tilde{m} = \tilde{n} \quad \tilde{q} \in \tilde{n} = \tilde{o} \\
& \quad \text{trans } \tilde{p} \tilde{q} \in \tilde{m} = \tilde{o}
\end{align*}
\]
Properties of $\equiv$

\[ \text{let } \tilde{n} \in \text{List}^\infty \]

\[ \text{refl } \tilde{n} \in \tilde{n} \equiv \tilde{n} \]

\[ \text{refl } \text{Nil}^\infty \quad = \text{EqNil}^\infty \]

\[ \text{refl } (\text{Cons}^\infty n \ \tilde{n}) \equiv \text{EqCons}^\infty (\text{refl } n) (\text{refl } \tilde{n}) \]

\[ \text{let } \tilde{p} \in \tilde{m} = \tilde{n} \quad \tilde{q} \in \tilde{n} = \tilde{o} \]

\[ \text{trans } \tilde{p} \tilde{q} \in \tilde{m} = \tilde{o} \]

\[ \text{trans } \text{EqNil}^\infty \quad \text{EqNil}^\infty \equiv \text{EqNil}^\infty \]

\[ \text{trans } (\text{EqCons}^\infty p \ \tilde{p}) (\text{EqCons}^\infty q \ \tilde{q}) \]

\[ = \text{EqCons}^\infty (\text{trans } p \ q) (\text{trans } \tilde{p} \tilde{q}) \]
A simple proof

\[
\text{let } \quad n \in \text{Nat} \\
\text{lem } n \in \text{mapS} \ (\text{from } n) = \text{from} \ (n + 1)
\]
A simple proof

\[
\text{let } \quad n \in \text{Nat} \\
\text{lem } n \in \text{mapS (from } n \text{)} = \text{from (} n + 1 \text{)} \\
\text{lem } n = \text{EqCons}^\infty (n + 1) (\text{lem (} n + 1 \text{)})
\]
A simple proof

\[
\begin{align*}
&\text{let } n \in \text{Nat} \\
&\quad \text{lem } n \in \text{map} S \ (\text{from } n) = \text{from } (n + 1) \\
&\quad \text{lem } n = \text{EqCons}^\infty (n + 1) \ (\text{lem } (n + 1)) \\
\end{align*}
\]

Coinductive reasoning can be easy.
A simple proof

let

\[ \frac{n \in \textbf{Nat}}{\text{lem } n \in \text{map} S \ (\text{from } n) = \text{from } (n + 1)} \]

\[ \text{lem } n = \text{EqCons}^\infty (n + 1) (\text{lem } (n + 1)) \]

- Coinductive reasoning can be easy.
- Guarded coinduction is guarded corecursion on proofs.
A simple proof

\[
\begin{align*}
\text{let} & \quad n \in \text{Nat} \\
\text{lem } n & \in \text{mapS} \ (\text{from } n) = \text{from } (n + 1) \\
\text{lem } n & = \text{EqCons}^\omega (n + 1) \ (\text{lem } (n + 1))
\end{align*}
\]

- Coinductive reasoning can be easy.
- Guarded coinduction is guarded corecursion on proofs.
- There is no need to construct bisimulations.
The mirror

Functions are codata. Consumer contract: You may only apply a function.
The mirror

<table>
<thead>
<tr>
<th>data</th>
<th>codata</th>
</tr>
</thead>
</table>

Where do functions live?

Functions are codata.

Consumer contract:
You may only apply a function.
The mirror

<table>
<thead>
<tr>
<th>data</th>
<th>codata</th>
</tr>
</thead>
<tbody>
<tr>
<td>inductive</td>
<td></td>
</tr>
</tbody>
</table>
The mirror

<table>
<thead>
<tr>
<th>data</th>
<th>codata</th>
</tr>
</thead>
<tbody>
<tr>
<td>inductive</td>
<td>coinductive</td>
</tr>
</tbody>
</table>
### The mirror

<table>
<thead>
<tr>
<th>data</th>
<th>codata</th>
</tr>
</thead>
<tbody>
<tr>
<td>inductive</td>
<td>coinductive</td>
</tr>
<tr>
<td>finite objects</td>
<td></td>
</tr>
</tbody>
</table>
The mirror

<table>
<thead>
<tr>
<th>data</th>
<th>codata</th>
</tr>
</thead>
<tbody>
<tr>
<td>inductive</td>
<td>coinductive</td>
</tr>
<tr>
<td>finite objects</td>
<td>infinite objects</td>
</tr>
</tbody>
</table>
## The mirror

<table>
<thead>
<tr>
<th>data</th>
<th>codata</th>
</tr>
</thead>
<tbody>
<tr>
<td>inductive</td>
<td>coinductive</td>
</tr>
<tr>
<td>finite objects</td>
<td>infinite objects</td>
</tr>
<tr>
<td>structural recursion</td>
<td></td>
</tr>
</tbody>
</table>

Where do functions live?

Functions are codata.

Consumer contract:

You may only apply a function.
## The mirror

<table>
<thead>
<tr>
<th>data</th>
<th>codata</th>
</tr>
</thead>
<tbody>
<tr>
<td>inductive</td>
<td>coinductive</td>
</tr>
<tr>
<td>finite objects</td>
<td>infinite objects</td>
</tr>
<tr>
<td>structural recursion</td>
<td>guarded corecursion</td>
</tr>
</tbody>
</table>
### The mirror

<table>
<thead>
<tr>
<th>data</th>
<th>codata</th>
</tr>
</thead>
<tbody>
<tr>
<td>inductive</td>
<td>coinductive</td>
</tr>
<tr>
<td>finite objects</td>
<td>infinite objects</td>
</tr>
<tr>
<td>structural recursion</td>
<td>guarded corecursion</td>
</tr>
<tr>
<td>structural induction</td>
<td></td>
</tr>
</tbody>
</table>

Functions are codata. Consumer contract: You may only apply a function.
The mirror

<table>
<thead>
<tr>
<th>data</th>
<th>codata</th>
</tr>
</thead>
<tbody>
<tr>
<td>inductive</td>
<td>coinductive</td>
</tr>
<tr>
<td>finite objects</td>
<td>infinite objects</td>
</tr>
<tr>
<td>structural recursion</td>
<td>guarded corecursion</td>
</tr>
<tr>
<td>structural induction</td>
<td>guarded coinduction</td>
</tr>
</tbody>
</table>

Where do functions live?

Functions are codata.

Consumer contract:

You may only apply a function.
The mirror

<table>
<thead>
<tr>
<th>data</th>
<th>codata</th>
</tr>
</thead>
<tbody>
<tr>
<td>inductive</td>
<td>coinductive</td>
</tr>
<tr>
<td>finite objects</td>
<td>infinite objects</td>
</tr>
<tr>
<td>structural recursion</td>
<td>guarded corecursion</td>
</tr>
<tr>
<td>structural induction</td>
<td>guarded coinduction</td>
</tr>
</tbody>
</table>

Where do functions live?
The mirror

<table>
<thead>
<tr>
<th>data</th>
<th>codata</th>
</tr>
</thead>
<tbody>
<tr>
<td>inductive</td>
<td>coinductive</td>
</tr>
<tr>
<td>finite objects</td>
<td>infinite objects</td>
</tr>
<tr>
<td>structural recursion</td>
<td>guarded corecursion</td>
</tr>
<tr>
<td>structural induction</td>
<td>guarded coinduction</td>
</tr>
</tbody>
</table>

Where do functions live?
- Functions are codata.
The mirror

<table>
<thead>
<tr>
<th>data</th>
<th>codata</th>
</tr>
</thead>
<tbody>
<tr>
<td>inductive</td>
<td>coinductive</td>
</tr>
<tr>
<td>finite objects</td>
<td>infinite objects</td>
</tr>
<tr>
<td>structural recursion</td>
<td>guarded corecursion</td>
</tr>
<tr>
<td>structural induction</td>
<td>guarded coinduction</td>
</tr>
</tbody>
</table>

Where do functions live?
Functions are codata.
Consumer contract:
You may only apply a function.
Leibniz ?

\[
\begin{align*}
\text{let} & \quad P \in \text{Nat} \rightarrow \text{Type} \quad q \in m = n \quad m \in \text{List} \quad p \in P \quad m \\
\text{leibniz} & \quad P \quad p \in P \quad n
\end{align*}
\]
Leibniz?

\[ P \in \textbf{Nat} \rightarrow \textbf{Type} \quad \bar{q} \in \bar{m} = \bar{n} \quad \bar{m} \in \textbf{List} \quad p \in P \bar{m} \]

\[
\text{let } \quad \text{leibniz } P \quad \bar{p}^ \top p \in P \bar{n} \]

\[
\text{leibniz } P \quad \text{EqNil} \quad \text{Nil} \quad p = p
\]

\[
\text{leibniz } P \quad (\text{EqCons } q \bar{q}) \quad (\text{Cons } m \bar{m}) \quad p =
\]

\[
\text{leibniz } (\lambda n \rightarrow P \quad (\text{Cons } n \bar{m})) \quad m q
\]

\[
(\text{leibniz } (\lambda \bar{n} \rightarrow P \quad (\text{Cons } m \bar{n})) \quad \bar{m} \bar{q} \bar{p})
\]
Leibniz ?

\[ P \in \text{Nat} \rightarrow \text{Type} \quad \bar{q} \in \bar{m} = \bar{n} \quad \bar{m} \in \text{List} \quad p \in P \bar{m} \]

\[ \text{leibniz } P \quad p \in P \bar{n} \]

\[ \text{leibniz } P \text{ EqNil} \quad \text{Nil} \quad p = p \]

\[ \text{leibniz } P \ (\text{EqCons } q \ \bar{q}) \ (\text{Cons } m \ \bar{m}) \ p = \]

\[ \text{leibniz } (\lambda n \rightarrow P \ (\text{Cons } n \ \bar{m})) \ m \ q \]

\[ (\text{leibniz } (\lambda \bar{n} \rightarrow P \ (\text{Cons } m \ \bar{n})) \ \bar{m} \ \bar{q} \ p) \]

\[ \text{leibniz doesn’t dualize to } \text{List}^\infty. \]
Observational Type Theory

We can implement by internalizing the setoid model – see my LICS 99 paper Extensional Type Theory, intensionally. Using this construction we implement both consumer and producer contracts without giving up decidability. This is based on a translation of Observational Type Theory into intensional Type Theory + a proof irrelevant universe of propositions.

Alternative: any two hypothetical proofs of are convertible.
Observational Type Theory

We can implement \textit{leibniz} by internalizing the setoid model – see my LICS 99 paper \textit{Extensional Type Theory, intensionally}. 
Observational Type Theory

We can implement Leibniz by internalizing the setoid model – see my LICS 99 paper *Extensional Type Theory, intensionally.*

Using this construction we implement both consumer and producer contracts without giving up decidability.
Observational Type Theory

We can implement \( \text{leibniz} \) by internalizing the setoid model – see my LICS 99 paper *Extensional Type Theory, intensionally*.

Using this construction we implement both consumer and producer contracts without giving up decidability.

This is based on a translation of Observational Type Theory into intensional Type Theory + a proof irrelevant universe of propositions.
Observational Type Theory

We can implement \textit{leibniz} by internalizing the setoid model – see my LICS 99 paper \textit{Extensional Type Theory, intensionally}.

Using this construction we implement both consumer and producer contracts without giving up decidability.

This is based on a translation of Observational Type Theory into intensional Type Theory + a proof irrelevant universe of propositions.

Alternative: any two hypothetical proofs of \texttt{False} are convertible.
A short history of Type Theory

No contracts, not even producer contracts. Instead of
we write

Impredicative encodings of data

Codata – p.17
A short history of Type Theory

Anarchy
Anarchy

No contracts, not even producer contracts.
A short history of Type Theory

Anarchy

No contracts, not even producer contracts. Instead of \( \prod n \in \text{Nat}: \ldots \) we write
\( \prod n \in \text{Nat}. (\text{Ind } n) \rightarrow \ldots \)
A short history of Type Theory

Anarchy

No contracts, not even producer contracts. Instead of $\prod n \in \text{Nat} : \ldots$ we write $\prod n \in \text{Nat}.(\text{Ind } n) \rightarrow \ldots$. 

Impredicative encodings of data
A short history of Type Theory

Wild West
A short history of Type Theory

Wild West
Producer contracts but no consumer contracts.
A short history of Type Theory

Wild West
Producer contracts but no consumer contracts.
We can quantify over $\mathbb{Nat}$
A short history of Type Theory

Wild West
Producer contracts but no consumer contracts.
We can quantify over \( \mathbb{Nat} \)
We have to verify again and again that a consumer of codata respects equality.
A short history of Type Theory

Wild West

Producer contracts but no consumer contracts.
We can quantify over $\mathbb{Nat}$
We have to verify again and again that a consumer of codata respects equality.

*Intensional Type Theory*
A short history of Type Theory

Rule of law
A short history of Type Theory

Rule of law
Producer and consumer contracts.
A short history of Type Theory

Rule of law
Producer and consumer contracts.
We can quantify over $\mathbb{Nat}$
A short history of Type Theory

Rule of law
Producer and consumer contracts.
We can quantify over $\mathbb{Nat}$
We know that any consumer of codata respects equality.
A short history of Type Theory

Rule of law
Producer and consumer contracts.
We can quantify over $\text{Nat}$
We know that any consumer of codata respects equality.

*Observational Type Theory*
The goal of our recently funded EPSRC project Decidable Type Theory with Observational Equality is to implement a Type Theory with observational equality (Observational Epigram). We want to improve on my LICS 99 paper by adding the conversion equality and hence strictly extend intensional Type Theory. We also want to realize another extension of the mirror: data codata subsettypes quotienttypes.
The goal of our recently funded EPSRC project *Decidable Type Theory with Observational Equality* is to implement a Type Theory with observational equality (Observational Epigram)
The goal of our recently funded EPSRC project *Decidable Type Theory with Observational Equality* is to implement a Type Theory with observational equality (Observational Epigram).

We want to improve on my LICS 99 paper by adding the conversion equality

\[ \text{leibniz \ldots refl } x \equiv x \]
Observational Epigram

The goal of our recently funded EPSRC project *Decidable Type Theory with Observational Equality* is to implement a Type Theory with observational equality (Observational Epigram)

We want to improve on my LICS 99 paper by adding the conversion equality

\[ \text{leibniz} \ldots \text{refl } x \equiv x \]

And hence strictly extend intensional Type Theory.
The goal of our recently funded EPSRC project *Decidable Type Theory with Observational Equality* is to implement a Type Theory with observational equality (Observational Epigram).

We want to improve on my LICS 99 paper by adding the conversion equality

\[ \text{leibniz} \ldots \text{refl } x \equiv x \]

And hence strictly extend intensional Type Theory.

We also want to realize another extension of the mirror:
The goal of our recently funded EPSRC project *Decidable Type Theory with Observational Equality* is to implement a Type Theory with observational equality (Observational Epigram).

We want to improve on my LICS 99 paper by adding the conversion equality

\[ \text{leibniz} \ldots \text{refl } x \equiv x \]

And hence strictly extend intensional Type Theory.

We also want to realize another extension of the mirror:

<table>
<thead>
<tr>
<th>data</th>
<th>codata</th>
</tr>
</thead>
<tbody>
<tr>
<td>subset types</td>
<td>quotient types</td>
</tr>
</tbody>
</table>