Universes of data types in constructive type theory

Lecture 2: Inductive data types, generically

Fredrik Nordvall Forsberg

University of Strathclyde
https://fredriknf.com/pc22/

Proof and Computation 2022 Autumn School, Fischbachau

Examples of inductive definitions

Martin-Löf (1972, 1979, 1980, . . .)

We have seen examples of inductive definitions such as $\mathbb{N}$ and lists.

Similarly the first accounts of Martin-Löf type theory included specific inductive definitions:

- $\mathbb{N}$, finite sets (1972)
- W-types (1979)
- Kleene's $\mathcal{O}$, lists (1980)

The system is considered open; new inductive types should be added as needed.

> *"We can follow the same pattern used to define natural numbers to introduce other inductively defined sets. We see here the example of lists." – Martin-Löf 1980*

Church encodings

Pfenning and Paulin-Mohring (1989)

- First attempt in Calculus of Constructions: use Church encodings of inductive types.
- E.g.

$$\mathbb{N} := (X : \mathsf{Type}) \to X \to (X \to X) \to X$$

$$a =_A b := (X : A \to \mathsf{Type}) \to X(a) \to X(b)$$

Church encodings

Pfenning and Paulin-Mohring (1989)

- First attempt in Calculus of Constructions: use Church encodings of inductive types.
- E.g.

$$\mathbb{N} := (X : \mathrm{Type}) \to X \to (X \to X) \to X$$

$$a =_A b := (X : A \to \mathrm{Type}) \to X(a) \to X(b)$$

- Problems:
  - Uses impredicativity in an essential way.
  - Induction (dependent elimination) is not derivable in CoC for any encoding [Geuvers 2001]. (Can be corrected using a refined construction; see Awodey, Frey and Speight [2018].)

Church encodings

Pfenning and Paulin-Mohring (1989)

- First attempt in Calculus of Constructions: use Church encodings of inductive types.
- E.g.

$$\mathbb{N} := (X : \mathsf{Type}) \to X \to (X \to X) \to X : \mathsf{Type}$$

$$a =_A b := (X : A \to \mathsf{Type}) \to X(a) \to X(b) : \mathsf{Type}$$

- Problems:
  - Uses impredicativity in an essential way.
  - Induction (dependent elimination) is not derivable in CoC for any encoding [Geuvers 2001]. (Can be corrected using a refined construction; see Awodey, Frey and Speight [2018].)

Church encodings

Pfenning and Paulin-Mohring (1989)

- First attempt in Calculus of Constructions: use Church encodings of inductive types.
- E.g.

$$\mathbb{N} := (X : \mathsf{Type}) \to X \to (X \to X) \to X : \mathsf{Type}$$

$$a =_A b := (X : A \to \mathsf{Type}) \to X(a) \to X(b) : \mathsf{Type}$$

- Problems:
  - Uses impredicativity in an essential way.
  - Induction (dependent elimination) is not derivable in CoC for any encoding [Geuvers 2001]. (Can be corrected using a refined construction; see Awodey, Frey and Speight [2018].)

Church encodings

Pfenning and Paulin-Mohring (1989)

- First attempt in Calculus of Constructions: use Church encodings of inductive types.
- E.g.

$$\mathbb{N} := (X : \mathrm{Type}) \to X \to (X \to X) \to X : \mathrm{Type}$$

$$a =_A b := (X : A \to \mathrm{Type}) \to X(a) \to X(b) : \mathrm{Type}$$

- Problems:
  - Uses impredicativity in an essential way.
  - Induction (dependent elimination) is not derivable in CoC for any encoding [Geuvers 2001]. (Can be corrected using a refined construction; see Awodey, Frey and Speight [2018].)
- Solution: Calculus of Inductive Constructions with inductive types builtin (according to schema).

Syntactic schemata

Backhouse (1987), Coquand and Paulin-Mohring (1990), Dybjer (1994), ...

Dybjer (1994) considers constructors of the form

$$\begin{aligned}\mathsf{intro}_D : {} & (A :: \sigma) \\ & (b :: \beta[A]) \to \\ & (u :: \gamma[A, b]) \to \\ & D\end{aligned}$$

where

- σ is a sequence of types for parameters [‘$x :: Y$’ telescope notation]
- $\beta[A]$ is a sequence of types for non-inductive arguments.
- $\gamma[A, b]$ is a sequence of types for inductive arguments:
  - Each $\gamma_i[A, b]$ is of the form $\xi_i[A, b] \to D$ (strict positivity).

Syntactic schemata (cont.)

- The elimination and computation rules are determined by an inversion principle.
- Infinite axiomatisation.
- Inprecise; '$\ldots$' everywhere.
- No way to reason about an arbitrary inductive definition *inside* the system (generic map etc.).

Syntax internalised

Dybjer and Setzer (1999, 2003, 2006) [for IR], Morris, Altenkirch and McBride (2007) ...

- Setzer wanted to analyse the proof-theoretical strength of Dybjer's schema version of induction-recursion.
- Hard with lots of '...' around...
- So they developed an axiomatisation where the syntax has been internalised into the system.
- Basic idea (simplified for inductive definitions) : the type is "given by constructors", so describe the domain of the constructor

$$\mathsf{intro}_{D_\gamma} : \mathsf{Arg}(\gamma, D_\gamma) \to D_\gamma$$

[γ is "code" that contains the necessary information to describe D_γ.]

Basic idea in some more detail

- Universe SP of codes for the domain of constructors of inductively defined sets. [SP stands for Strictly Positive.]
- Decoding function $\mathrm{Arg} : \mathrm{SP} \to \mathrm{Type} \to \mathrm{Type}$. [$\mathrm{Arg}(\gamma, X)$ is the domain where X is used for the inductive arguments.]
- For every $\gamma : \mathrm{SP}$, stipulate that there is a set D_γ and a constructor $\mathrm{intro}_\gamma : \mathrm{Arg}(\gamma, D_\gamma) \to D_\gamma$.
- Calculate types for elimination and computation rules.

Underlying type theory ("logical framework")

We assume we have the following types:

- Dependent function types $(x : A) \to B$
- Dependent pair types $(x : A) \times B$
- A unit type $\mathbf{1}$, and a type of Booleans Bool
- (For future use: identity types $a =_A a'$)

Underlying type theory (“logical framework”)

We assume we have the following types:

- Dependent function types $(x : A) \to B$
- Dependent pair types $(x : A) \times B$
- A unit type $\mathbf{1}$, and a type of Booleans Bool
- (For future use: identity types $a =_A a'$)

The rest we will add generically!

Idea for SP

Inductive types are determined by their constructors, so analyse possible constructors.

For example:

$$\_ :: \_ : (x : A) \to (xs : \mathsf{List}\, A) \to \mathsf{List}\, A$$

Idea for SP

Inductive types are determined by their constructors, so analyse possible constructors.

For example:

$$\_ :: \_ : (x : A) \to (xs : \mathrm{List}\, A) \to \mathrm{List}\, A$$

- $x : A$ is a *non-inductive argument* (later arguments could depend on it)
- $xs : \mathrm{List}\, A$ is an *inductive argument* (could also have infinite arity)

Idea for SP

Inductive types are determined by their constructors, so analyse possible constructors.

For example:

$$\_ :: \_ : (x : A) \to (xs : \mathsf{List}\, A) \to \mathsf{List}\, A$$

- $x : A$ is a *non-inductive argument* (later arguments could depend on it)
- $xs : \mathsf{List}\, A$ is an *inductive argument* (could also have infinite arity)

Constructor for well-founded trees $\mathsf{W}\, S\, P$:

$$\mathsf{sup} : (s : S) \to (f : P[s] \to \mathsf{W}\, S\, P) \to \mathsf{W}\, S\, P$$

The universe SP of codes

Fix two universes (U_{sc}, T_{sc}) and (U_{ar}, T_{ar}). We will draw side conditions (non-inductive arguments) and arities of inductive arguments from these.

Formation

$$\frac{}{\mathsf{SP}\ \text{type}}$$

Introduction

$$\frac{}{\mathsf{done} : \mathsf{SP}}$$

$$\frac{A : U_{sc} \qquad \gamma : T_{sc}(A) \to \mathsf{SP}}{\mathsf{nonind}\, A\, \gamma : \mathsf{SP}} \qquad \frac{A : U_{ar} \qquad \gamma : \mathsf{SP}}{\mathsf{ind}\, A\, \gamma : \mathsf{SP}}$$

Elimination, computation ...

By changing U_{sc} and U_{ar}, we can restrict to different subclasses of inductive definitions.

Subclasses of inductive definitions

Subclasses of inductive definitions

Discrete types:

- $U_{sc} = \{\text{discrete types}\}$, $U_{ar} = \{\text{unit type}\}$

Subclasses of inductive definitions

Discrete types:

- $U_{sc} = \{\text{discrete types}\}$, $U_{ar} = \{\text{unit type}\}$

Propositional types:

- $U_{sc} = \{\text{propositional types}\}$, $U_{ar} = \{\text{arbitrary types}\}$

Finite types:

- $U_{sc} = \{\text{finite types}\}$, $U_{ar} = \{\text{finite types}\}$

And so on.

Arg and D_γ

Codes are given their meaning by $\mathrm{Arg} : \mathrm{SP} \to \mathrm{Type} \to \mathrm{Type}$.

$$
\begin{aligned}
\mathit{Arg}\, \mathrm{done}\, X &\equiv \mathbf{1} \\
\mathit{Arg}\, (\mathrm{nonind}\, A\, \gamma)\, X &\equiv (y : T_{sc}(A)) \times (\mathrm{Arg}\, (\gamma y)\, X) \\
\mathit{Arg}\, (\mathrm{ind}\, A\, \gamma)\, X &\equiv (T_{ar}(A) \to X) \times (\mathit{Arg}\, \gamma\, X)
\end{aligned}
$$

Arg and D_γ

Codes are given their meaning by Arg : SP $\to$ Type $\to$ Type.

$$\begin{aligned} \mathit{Arg}\,\mathrm{done}\,X &\equiv \mathbf{1} \\ \mathit{Arg}\,(\mathrm{nonind}\,A\,\gamma)\,X &\equiv (y : T_{sc}(A)) \times (\mathrm{Arg}\,(\gamma y)\,X) \\ \mathit{Arg}\,(\mathrm{ind}\,A\,\gamma)\,X &\equiv (T_{ar}(A) \to X) \times (\mathit{Arg}\,\gamma\,X) \end{aligned}$$

One generic inductive definition (parametrised by γ):

Formation

$$\frac{\gamma : \mathrm{SP}}{D_\gamma \ \mathrm{type}}$$

Introduction

$$\frac{x : \mathrm{Arg}\,\gamma\,D_\gamma}{\mathrm{intro}_\gamma\,x : D_\gamma}$$

Multiple constructors

Assuming U_{sc} contains Bool, we can encode two constructors into one:

$$\gamma +_{\mathsf{SP}} \psi :\equiv \mathsf{nonind}(\mathsf{Bool}, \lambda x.\ \mathsf{if}\ x\ \mathsf{then}\ \gamma\ \mathsf{else}\ \psi)$$

The point being:

$$\mathsf{Arg}\,(\gamma +_{\mathsf{SP}} \psi)\, X \cong (\mathsf{Arg}\,\gamma\, X) + (\mathsf{Arg}\,\psi\, X)$$

and $(A + B \to C) \cong (A \to C) \times (B \to C)$.

Example: the code for List A

We have

$$\gamma_{\mathsf{List}\,A} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{nonind}(A, \lambda\_.\mathsf{ind}(\mathbf{1}, \mathsf{done}))$$

with $\mathsf{List}\,A \equiv D_{\gamma_{\mathsf{List}\,A}}$.

$[] : \mathsf{List}\,A$

$[] \equiv \boxed{?_0 : D_{\gamma_{\mathsf{List}\,A}}}$

$\_ :: \_ : A \to \mathsf{List}\,A \to \mathsf{List}\,A$

$x :: xs \equiv \boxed{?_1 : D_{\gamma_{\mathsf{List}\,A}}}$

Example: the code for List A

We have

$$\gamma_{\mathsf{List}\,A} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{nonind}(A, \lambda\_.\mathsf{ind}(\mathbf{1}, \mathsf{done}))$$

with List $A \equiv D_{\gamma_{\mathsf{List}\,A}}$.

$[] : \mathsf{List}\,A$

$[] \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}\ \boxed{?_2 : \mathsf{Arg}\,\gamma_{\mathsf{List}\,A}\,D_{\gamma_{\mathsf{List}\,A}}}$

$\_ :: \_ : A \to \mathsf{List}\,A \to \mathsf{List}\,A$

$x :: xs \equiv \boxed{?_1 : D_{\gamma_{\mathsf{List}\,A}}}$

Example: the code for List A

We have

$$\gamma_{\mathsf{List}\,A} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{nonind}(A, \lambda\_.\mathsf{ind}(\mathbf{1}, \mathsf{done}))$$

with List $A \equiv D_{\gamma_{\mathsf{List}\,A}}$.

$[] : \mathsf{List}\,A$

$[] \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}\ \boxed{?_2 : (x : \mathsf{Bool}) \times (\text{if } x \text{ then } \mathbf{1} \text{ else } A \times (\mathbf{1} \to \mathsf{List}\,A) \times \mathbf{1})}$

$\_ :: \_ : A \to \mathsf{List}\,A \to \mathsf{List}\,A$

$x :: xs \equiv \boxed{?_1 : D_{\gamma_{\mathsf{List}\,A}}}$

Example: the code for List A

We have

$$\gamma_{\mathsf{List}\,A} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{nonind}(A, \lambda\_.\mathsf{ind}(\mathbf{1}, \mathsf{done}))$$

with $\mathsf{List}\,A \equiv D_{\gamma_{\mathsf{List}\,A}}$.

$[] : \mathsf{List}\,A$

$[] \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}(\mathsf{tt}, \boxed{?_3 : \mathsf{if}\ \mathsf{tt}\ \mathsf{then}\ \mathbf{1}\ \mathsf{else}\ A \times (\mathbf{1} \to \mathsf{List}\,A) \times \mathbf{1}})$

$\_ :: \_ : A \to \mathsf{List}\,A \to \mathsf{List}\,A$

$x :: xs \equiv \boxed{?_1 : D_{\gamma_{\mathsf{List}\,A}}}$

Example: the code for List A

We have

$$\gamma_{\mathsf{List}\,A} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{nonind}(A, \lambda\_.\mathsf{ind}(\mathbf{1}, \mathsf{done}))$$

with $\mathsf{List}\,A \equiv D_{\gamma_{\mathsf{List}\,A}}$.

$[] : \mathsf{List}\,A$

$[] \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}(\mathsf{tt}, \boxed{?_3 : \mathbf{1}})$

$\_ :: \_ : A \to \mathsf{List}\,A \to \mathsf{List}\,A$

$x :: xs \equiv \boxed{?_1 : D_{\gamma_{\mathsf{List}\,A}}}$

Example: the code for List A

We have

$$\gamma_{\mathsf{List}\,A} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{nonind}(A, \lambda\_.\mathsf{ind}(\mathbf{1}, \mathsf{done}))$$

with $\mathsf{List}\,A \equiv D_{\gamma_{\mathsf{List}\,A}}$.

$[] : \mathsf{List}\,A$

$[] \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}(\mathsf{tt}, \star)$

$\_ :: \_ : A \to \mathsf{List}\,A \to \mathsf{List}\,A$

$x :: xs \equiv \boxed{?_1 : D_{\gamma_{\mathsf{List}\,A}}}$

Example: the code for List A

We have

$$\gamma_{\mathsf{List}\,A} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{nonind}(A, \lambda\_.\mathsf{ind}(\mathbf{1}, \mathsf{done}))$$

with $\mathsf{List}\,A \equiv D_{\gamma_{\mathsf{List}\,A}}$.

$[] : \mathsf{List}\,A$

$[] \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}\,(\mathsf{tt}, \star)$

$\_ :: \_ : A \to \mathsf{List}\,A \to \mathsf{List}\,A$

$x :: xs \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}\ \boxed{?_4 : \mathsf{Arg}\,\gamma_{\mathsf{List}\,A}\,D_{\gamma_{\mathsf{List}\,A}}}$

Example: the code for List A

We have

$$\gamma_{\mathsf{List}\,A} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{nonind}(A, \lambda\_.\mathsf{ind}(\mathbf{1}, \mathsf{done}))$$

with $\mathsf{List}\,A \equiv D_{\gamma_{\mathsf{List}\,A}}$.

$[] : \mathsf{List}\,A$

$[] \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}(\mathsf{tt}, \star)$

$\_ :: \_ : A \to \mathsf{List}\,A \to \mathsf{List}\,A$

$x :: xs \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}$ $?_4 : (x : \mathsf{Bool}) \times (\text{if } x \text{ then } \mathbf{1} \text{ else } A \times (\mathbf{1} \to \mathsf{List}\,A) \times$

Example: the code for List A

We have

$$\gamma_{\mathsf{List}\,A} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{nonind}(A, \lambda\_.\mathsf{ind}(\mathbf{1}, \mathsf{done}))$$

with $\mathsf{List}\,A \equiv D_{\gamma_{\mathsf{List}\,A}}$.

$[] : \mathsf{List}\,A$

$[] \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}(\mathsf{tt}, \star)$

$\_ :: \_ : A \to \mathsf{List}\,A \to \mathsf{List}\,A$

$x :: xs \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}(\mathsf{ff}, \; ?_5 : A \times (\mathbf{1} \to \mathsf{List}\,A) \times \mathbf{1}\;)$

Example: the code for List A

We have

$$\gamma_{\mathsf{List}\,A} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{nonind}(A, \lambda\_.\mathsf{ind}(\mathbf{1}, \mathsf{done}))$$

with $\mathsf{List}\,A \equiv D_{\gamma_{\mathsf{List}\,A}}$.

$[] : \mathsf{List}\,A$

$[] \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}(\mathsf{tt}, \star)$

$\_ :: \_ : A \to \mathsf{List}\,A \to \mathsf{List}\,A$

$x :: xs \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}(\mathsf{ff}, (\,?_6 : A\,,\ ?_6 : \mathbf{1} \to \mathsf{List}\,A\,, \star))$

Example: the code for List A

We have

$$\gamma_{\mathsf{List}\,A} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{nonind}(A, \lambda\_.\mathsf{ind}(\mathbf{1}, \mathsf{done}))$$

with $\mathsf{List}\,A \equiv D_{\gamma_{\mathsf{List}\,A}}$.

$[] : \mathsf{List}\,A$

$[] \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}\,(\mathsf{tt}, \star)$

$\_ :: \_ : A \to \mathsf{List}\,A \to \mathsf{List}\,A$

$x :: xs \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}\,(\mathsf{ff}, (x, \boxed{?_6 : \mathbf{1} \to \mathsf{List}\,A}, \star))$

Example: the code for List A

We have

$$\gamma_{\mathsf{List}\,A} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{nonind}(A, \lambda\_.\mathsf{ind}(\mathbf{1}, \mathsf{done}))$$

with $\mathsf{List}\,A \equiv D_{\gamma_{\mathsf{List}\,A}}$.

$[] : \mathsf{List}\,A$

$[] \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}(\mathsf{tt}, \star)$

$\_ :: \_ : A \to \mathsf{List}\,A \to \mathsf{List}\,A$

$x :: xs \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}(\mathsf{ff}, (x, \lambda\_.\ \boxed{?_7 : \mathsf{List}\,A}\ , \star))$

Example: the code for List A

We have

$$\gamma_{\mathsf{List}\,A} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{nonind}(A, \lambda\_.\mathsf{ind}(\mathbf{1}, \mathsf{done}))$$

with $\mathsf{List}\,A \equiv D_{\gamma_{\mathsf{List}\,A}}$.

$[] : \mathsf{List}\,A$

$[] \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}(\mathsf{tt}, \star)$

$\_ :: \_ : A \to \mathsf{List}\,A \to \mathsf{List}\,A$

$x :: xs \equiv \mathsf{intro}_{\gamma_{\mathsf{List}\,A}}(\mathsf{ff}, (x, \lambda\_.xs, \star))$

The type of induction hypothesis

To state elimination and computation rules, we need a little bit more machinery.

The type of induction hypothesis

To state elimination and computation rules, we need a little bit more machinery.

Intuitively, the elimination rule says "it is enough to prove $P(\mathrm{intro}_\gamma\, x)$ assuming P already holds for all substructures of x".

The type of induction hypothesis

To state elimination and computation rules, we need a little bit more machinery.

Intuitively, the elimination rule says "it is enough to prove $P(\mathsf{intro}_\gamma\, x)$ assuming P already holds for all substructures of x".

Hence we define

$$\mathsf{All} : (\gamma : \mathsf{SP}) \to (X \to \mathsf{Type}) \to (\mathsf{Arg}\,\gamma\, X \to \mathsf{Type})$$

lifting $P : X \to \mathsf{Type}$ to substructures:

The type of induction hypothesis

To state elimination and computation rules, we need a little bit more machinery.

Intuitively, the elimination rule says "it is enough to prove $P(\mathsf{intro}_\gamma\, x)$ assuming P already holds for all substructures of x".

Hence we define

$$\mathsf{All} : (\gamma : \mathsf{SP}) \to (X \to \mathsf{Type}) \to (\mathsf{Arg}\,\gamma\, X \to \mathsf{Type})$$

lifting $P : X \to \mathsf{Type}$ to substructures:

$$\begin{aligned}
&\mathsf{All}\,\mathsf{done}\, P\, _ \equiv \mathbf{1}\\
&\mathsf{All}\,(\mathsf{nonind}\, A\, \gamma)\, P\,(a, y) \equiv \mathsf{All}\,(\gamma\, a)\, P\, y\\
&\mathsf{All}\,(\mathsf{ind}\, A\, \gamma)\, P\,(g, y) \equiv ((x : T_{ar}\, A) \to P\,(g\, x)) \times \mathsf{All}\,\gamma\, P\, y
\end{aligned}$$

All acts on dependent functions

$$\begin{aligned}
&\mathsf{All}\,\mathsf{done}\,P\,_ \equiv \mathbf{1}\\
&\mathsf{All}\,(\mathsf{nonind}\,A\,\gamma)\,P\,(a, y) \equiv \mathsf{All}\,(\gamma\,a)\,P\,y\\
&\mathsf{All}\,(\mathsf{ind}\,A\,\gamma)\,P\,(g, y) \equiv ((x : T_{ar}\,A) \to P\,(g\,x)) \times \mathsf{All}\,\gamma\,P\,y
\end{aligned}$$

All acts on dependent functions

$$
\begin{aligned}
&\mathsf{All}\,\mathsf{done}\,P\,\_ \equiv \mathbf{1}\\
&\mathsf{All}\,(\mathsf{nonind}\,A\,\gamma)\,P\,(a, y) \equiv \mathsf{All}\,(\gamma\,a)\,P\,y\\
&\mathsf{All}\,(\mathsf{ind}\,A\,\gamma)\,P\,(g, y) \equiv ((x : T_{ar}\,A) \to P\,(g\,x)) \times \mathsf{All}\,\gamma\,P\,y
\end{aligned}
$$

To state the computation rule, we also need the following fact: we can lift sections of P to sections of $\mathsf{All}\,\gamma\,P$.

All acts on dependent functions

$$
\begin{aligned}
&\mathrm{All}\,\mathrm{done}\,P\;\_\equiv \mathbf{1}\\
&\mathrm{All}\,(\mathrm{nonind}\,A\,\gamma)\,P\,(a, y) \equiv \mathrm{All}\,(\gamma\,a)\,P\,y\\
&\mathrm{All}\,(\mathrm{ind}\,A\,\gamma)\,P\,(g, y) \equiv ((x : T_{ar}\,A) \to P\,(g\,x)) \times \mathrm{All}\,\gamma\,P\,y
\end{aligned}
$$

To state the computation rule, we also need the following fact: we can lift sections of P to sections of $\mathrm{All}\,\gamma\,P$.

$$
\begin{aligned}
&\mathrm{every} : (\gamma : \mathrm{SP}) \to (f : (x : X) \to P\,x) \to ((y : \mathrm{Arg}\,\gamma\,X) \to \mathrm{All}\,\gamma\,P\,y)\\
&\mathrm{every}\,\mathrm{done}\,f\;\_\equiv \star\\
&\mathrm{every}\,(\mathrm{nonind}\,A\,\gamma)\,f\,(a, y) \equiv \mathrm{every}\,(\gamma\,a)\,f\,y\\
&\mathrm{every}\,(\mathrm{ind}\,A\,\gamma)\,f\,(g, y) \equiv (f \circ g, \mathrm{every}\,\gamma\,f\,y)
\end{aligned}
$$

All acts on dependent functions

$$\begin{aligned}
&\mathsf{All}\,\mathsf{done}\,P\,_ \equiv \mathbf{1}\\
&\mathsf{All}\,(\mathsf{nonind}\,A\,\gamma)\,P\,(a,y) \equiv \mathsf{All}\,(\gamma\,a)\,P\,y\\
&\mathsf{All}\,(\mathsf{ind}\,A\,\gamma)\,P\,(g,y) \equiv ((x : T_{ar}\,A) \to P\,(g\,x)) \times \mathsf{All}\,\gamma\,P\,y
\end{aligned}$$

To state the computation rule, we also need the following fact: we can lift sections of P to sections of $\mathsf{All}\,\gamma\,P$.

$$\begin{aligned}
&\mathsf{every} : (\gamma : \mathsf{SP}) \to (f : (x : X) \to P\,x) \to ((y : \mathsf{Arg}\,\gamma\,X) \to \mathsf{All}\,\gamma\,P\,y)\\
&\mathsf{every}\,\mathsf{done}\,f\,_ \equiv \star\\
&\mathsf{every}\,(\mathsf{nonind}\,A\,\gamma)\,f\,(a,y) \equiv \mathsf{every}\,(\gamma\,a)\,f\,y\\
&\mathsf{every}\,(\mathsf{ind}\,A\,\gamma)\,f\,(g,y) \equiv (f \circ g, \mathsf{every}\,\gamma\,f\,y)
\end{aligned}$$

Exercise: Can you make sense of All as a functor in the conventional sense? What categories are involved?

Elimination and computation rules

Formation

$$\frac{\gamma : \mathsf{SP}}{D_\gamma \ \mathsf{type}}$$

Introduction

$$\frac{x : \mathsf{Arg}\, \gamma\, D_\gamma}{\mathsf{intro}_\gamma\, x : D_\gamma}$$

Elimination and computation rules

Formation

$$\frac{\gamma : \mathsf{SP}}{D_\gamma \ \mathsf{type}}$$

Introduction

$$\frac{x : \mathsf{Arg}\,\gamma\, D_\gamma}{\mathsf{intro}_\gamma\, x : D_\gamma}$$

Elimination

$$\frac{z : D_\gamma \vdash C \ \mathsf{type} \quad x : \mathsf{Arg}\,\gamma\, D_\gamma, \bar{x} : \mathsf{All}\,\gamma\, C\, x \vdash d : C[z \mapsto \mathsf{intro}_\gamma\, x] \quad p : D_\gamma}{\mathsf{elim}_\gamma(C, d, p) : C[z \mapsto p]}$$

Computation

$$\mathsf{elim}_\gamma(C, d, \mathsf{intro}_\gamma\, a) \equiv d[x \mapsto a, \bar{x} \mapsto \mathsf{every}\,\gamma\,(\mathsf{elim}_\gamma(C, d))] : C[z \mapsto \mathsf{intro}_\gamma\, a]$$

Example: induction for natural numbers

For $\mathbb{N}$, we have $\gamma_{\mathbb{N}} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{ind}(\mathbf{1}, \mathsf{done})$.

Example: induction for natural numbers

For $\mathbb{N}$, we have $\gamma_{\mathbb{N}} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{ind}(\mathbf{1}, \mathsf{done})$.

Hence $\mathsf{Arg}\, \gamma_{\mathbb{N}} X \cong \mathbf{1} + X$.

We define $0 := \mathsf{intro}_{\gamma_{\mathbb{N}}}\,(\mathsf{inl}\, \star)$ and $\mathsf{suc}\, n := \mathsf{intro}_{\gamma_{\mathbb{N}}}\,(\mathsf{inr}\, n)$.

Example: induction for natural numbers

For $\mathbb{N}$, we have $\gamma_{\mathbb{N}} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{ind}(\mathbf{1}, \mathsf{done})$.

Hence $\mathsf{Arg}\, \gamma_{\mathbb{N}} X \cong \mathbf{1} + X$.

We define $0 :\equiv \mathsf{intro}_{\gamma_{\mathbb{N}}}\, (\mathsf{inl}\, \star)$ and $\mathsf{suc}\, n :\equiv \mathsf{intro}_{\gamma_{\mathbb{N}}}\, (\mathsf{inr}\, n)$.

Further

$$\begin{aligned} \mathsf{All}\, \gamma_{\mathbb{N}}\, P\, 0 &= \mathbf{1} \\ \mathsf{All}\, \gamma_{\mathbb{N}}\, P\, (\mathsf{suc}\, n) &\cong P\, n \end{aligned}$$

Example: induction for natural numbers

For $\mathbb{N}$, we have $\gamma_{\mathbb{N}} \equiv \mathsf{done} +_{\mathsf{SP}} \mathsf{ind}(\mathbf{1}, \mathsf{done})$.

Hence $\mathsf{Arg}\, \gamma_{\mathbb{N}} X \cong \mathbf{1} + X$.

We define $0 :\equiv \mathsf{intro}_{\gamma_{\mathbb{N}}}\, (\mathsf{inl}\, \star)$ and $\mathsf{suc}\, n :\equiv \mathsf{intro}_{\gamma_{\mathbb{N}}}\, (\mathsf{inr}\, n)$.

Further

$$\begin{aligned} \mathsf{All}\, \gamma_{\mathbb{N}}\, P\, 0 &= \mathbf{1} \\ \mathsf{All}\, \gamma_{\mathbb{N}}\, P\, (\mathsf{suc}\, n) &\cong P\, n \end{aligned}$$

As expected, the "step function"

$$x : \mathsf{Arg}\, \gamma\, D_\gamma, \bar{x} : \mathsf{All}\, \gamma\, C\, x \vdash d : C[z \mapsto \mathsf{intro}_\gamma\, x]$$

thus splits up into $d_0 : C[z \mapsto 0]$ and

$$n : \mathbb{N}, \bar{n} : C[z \mapsto n] \vdash d_{\mathsf{suc}} : C[z \mapsto \mathsf{suc}\, n]$$

and we recover the usual induction principle.

Does it make sense?

Does it make sense?

How do we know the proposed rules are sensible?

Does it make sense?

How do we know the proposed rules are sensible?

Could consider to prove "syntactic" theorems such as subject reduction, strong normalisation, decidability of type checking, etc.

Does it make sense?

How do we know the proposed rules are sensible?

Could consider to prove "syntactic" theorems such as subject reduction, strong normalisation, decidability of type checking, etc.

At the very least, we should be able to interpret the rules in (ideally a wide range of) models.

Does it make sense?

How do we know the proposed rules are sensible?

Could consider to prove “syntactic” theorems such as subject reduction, strong normalisation, decidability of type checking, etc.

At the very least, we should be able to interpret the rules in (ideally a wide range of) models.

Good first candidate: “naive” set-theoretic model.

Inductive definitions in classical set theory

Let us work in the closed-types-as-sets model:

$$
\begin{aligned}
[\![(x : A) \to B]\!] &= \prod_{x \in [\![A]\!]} [\![B]\!] \\
[\![(x : A) \times B]\!] &= \sum_{x \in [\![A]\!]} [\![B]\!] \\
[\![\mathbf{1}]\!] &= \{\star\} \\
[\![\mathsf{Bool}]\!] &= \{0, 1\} \\
[\![a =_A a']\!] &= \{\star \mid [\![a]\!] = [\![a']\!]\} \\
&\vdots
\end{aligned}
$$

Inductive definitions in classical set theory

Let us work in the closed-types-as-sets model:

$$\llbracket (x : A) \to B \rrbracket = \prod_{x \in \llbracket A \rrbracket} \llbracket B \rrbracket$$

$$\llbracket (x : A) \times B \rrbracket = \sum_{x \in \llbracket A \rrbracket} \llbracket B \rrbracket$$

$$\llbracket \mathbf{1} \rrbracket = \{\star\}$$

$$\llbracket \mathsf{Bool} \rrbracket = \{0, 1\}$$

$$\llbracket a =_A a' \rrbracket = \{\star \mid \llbracket a \rrbracket = \llbracket a' \rrbracket\}$$

$$\vdots$$

Ind. definitions represented by monotone operators $\Gamma : \mathsf{Set} \to \mathsf{Set}$.

For example: $\Gamma_{\mathbb{N}}(X) = \{0\} \cup \{\mathsf{suc}\, n \mid n \in X\}$.

Inductive definitions in classical set theory

Let us work in the closed-types-as-sets model:

$$\begin{aligned}
[\![(x : A) \to B]\!] &= \prod_{x \in [\![A]\!]} [\![B]\!] \\
[\![(x : A) \times B]\!] &= \sum_{x \in [\![A]\!]} [\![B]\!] \\
[\![\mathbf{1}]\!] &= \{\star\} \\
[\![\mathrm{Bool}]\!] &= \{0, 1\} \\
[\![a =_A a']\!] &= \{\star \mid [\![a]\!] = [\![a']\!]\} \\
&\vdots
\end{aligned}$$

Ind. definitions represented by monotone operators $\Gamma : \mathrm{Set} \to \mathrm{Set}$.

For example: $\Gamma_{\mathbb{N}}(X) = \{0\} \cup \{\mathrm{suc}\, n \mid n \in X\}$.

Inductive definition $I(\Gamma)$ interpreted as the result of iterating Γ:

$$\emptyset \subseteq \Gamma(\emptyset) \subseteq \Gamma^2(\emptyset) \subseteq \ldots$$

Inductive definitions in classical set theory

Let us work in the closed-types-as-sets model:

$$
\begin{aligned}
[\![(x : A) \to B]\!] &= \prod_{x \in [\![A]\!]} [\![B]\!] \\
[\![(x : A) \times B]\!] &= \sum_{x \in [\![A]\!]} [\![B]\!] \\
[\![\mathbf{1}]\!] &= \{\star\} \\
[\![\mathsf{Bool}]\!] &= \{0, 1\} \\
[\![a =_A a']\!] &= \{\star \mid [\![a]\!] = [\![a']\!]\} \\
&\vdots
\end{aligned}
$$

Ind. definitions represented by monotone operators $\Gamma : \mathsf{Set} \to \mathsf{Set}$.

For example: $\Gamma_{\mathbb{N}}(X) = \{0\} \cup \{\mathsf{suc}\, n \mid n \in X\}$.

Inductive definition $I(\Gamma)$ interpreted as the result of iterating Γ:

$$\emptyset \subseteq \Gamma(\emptyset) \subseteq \Gamma^2(\emptyset) \subseteq \ldots \subseteq \bigcup_{i < \omega} \Gamma^i(\emptyset)$$

Inductive definitions in classical set theory

Let us work in the closed-types-as-sets model:

$$\llbracket (x : A) \to B \rrbracket = \prod_{x \in \llbracket A \rrbracket} \llbracket B \rrbracket$$

$$\llbracket (x : A) \times B \rrbracket = \sum_{x \in \llbracket A \rrbracket} \llbracket B \rrbracket$$

$$\llbracket \mathbf{1} \rrbracket = \{\star\}$$

$$\llbracket \mathsf{Bool} \rrbracket = \{0, 1\}$$

$$\llbracket a =_A a' \rrbracket = \{\star \mid \llbracket a \rrbracket = \llbracket a' \rrbracket\}$$

$$\vdots$$

Ind. definitions represented by monotone operators $\Gamma : \mathsf{Set} \to \mathsf{Set}$.

For example: $\Gamma_{\mathbb{N}}(X) = \{0\} \cup \{\mathsf{suc}\, n \mid n \in X\}$.

Inductive definition $I(\Gamma)$ interpreted as the result of iterating Γ:

$$\emptyset \subseteq \Gamma(\emptyset) \subseteq \Gamma^2(\emptyset) \subseteq \ldots \subseteq \bigcup_{i<\omega} \Gamma^i(\emptyset) \subseteq \Gamma(\bigcup_{i<\omega} \Gamma^i(\emptyset)) \subseteq \ldots$$

Inductive definitions in classical set theory

Let us work in the closed-types-as-sets model:

$$
\begin{aligned}
[\![(x : A) \to B]\!] &= \prod_{x \in [\![A]\!]} [\![B]\!] \\
[\![(x : A) \times B]\!] &= \sum_{x \in [\![A]\!]} [\![B]\!] \\
[\![\mathbf{1}]\!] &= \{\star\} \\
[\![\mathsf{Bool}]\!] &= \{0, 1\} \\
[\![a =_A a']\!] &= \{\star \mid [\![a]\!] = [\![a']\!]\} \\
&\vdots
\end{aligned}
$$

Ind. definitions represented by monotone operators $\Gamma : \mathsf{Set} \to \mathsf{Set}$.

For example: $\Gamma_{\mathbb{N}}(X) = \{0\} \cup \{\mathsf{suc}\, n \mid n \in X\}$.

Inductive definition $I(\Gamma)$ interpreted as the result of iterating Γ:

$$\emptyset \subseteq \Gamma(\emptyset) \subseteq \Gamma^2(\emptyset) \subseteq \ldots \subseteq \bigcup_{i<\omega} \Gamma^i(\emptyset) \subseteq \Gamma(\bigcup_{i<\omega} \Gamma^i(\emptyset)) \subseteq \ldots$$

How do we know the process stops?

How do we know the process stops?

Easy case: If we know $\Gamma : \mathcal{P}(V) \to \mathcal{P}(V)$ for some set V, then $|V|$ must be an upper bound for the number of iterations needed.

How do we know the process stops?

Easy case: If we know $\Gamma : \mathcal{P}(V) \to \mathcal{P}(V)$ for some set V, then $|V|$ must be an upper bound for the number of iterations needed.

Alternatively, $I(\Gamma)$ can then be impredicatively constructed as the intersection of all Γ-closed subsets of A (cf. Church encodings).

How do we know the process stops?

Easy case: If we know $\Gamma : \mathcal{P}(V) \to \mathcal{P}(V)$ for some set V, then $|V|$ must be an upper bound for the number of iterations needed.

Alternatively, $I(\Gamma)$ can then be impredicatively constructed as the intersection of all Γ-closed subsets of A (cf. Church encodings).

More work: Say Γ is κ-based for cardinal κ if $x \in \Gamma(X)$ implies $x \in \Gamma(Y)$ for some $Y \subseteq X$ with $|Y| < \kappa$. (cf. Aczel 1977)

Example: $\Gamma_{\mathbb{N}} = X \mapsto \{0\} \cup \{\mathrm{suc}\, n \mid n \in X\}$ is 2-based.

Thm: If Γ is κ-based for a regular κ, then $I(\Gamma) = \Gamma^{\kappa}$.

Soundness of SP

Soundness of SP

It is not hard to show that $\mathrm{Arg}\,\gamma$ is monotone for each $\gamma : \mathsf{SP}$.

Soundness of SP

It is not hard to show that $\operatorname{Arg}\gamma$ is monotone for each $\gamma : \mathsf{SP}$.

Further each $\operatorname{Arg}\gamma$ is κ-based, if κ is greater than the cardinality of each set occurring as a side condition or an arity in γ. A regular κ' with this property always exists (using classical logic).

Soundness of SP

It is not hard to show that $\operatorname{Arg}\gamma$ is monotone for each $\gamma : \mathsf{SP}$.

Further each $\operatorname{Arg}\gamma$ is κ-based, if κ is greater than the cardinality of each set occurring as a side condition or an arity in γ. A regular κ' with this property always exists (using classical logic).

Hence $I(\operatorname{Arg}\gamma)$ exists.

Soundness of SP

It is not hard to show that $\mathrm{Arg}\,\gamma$ is monotone for each $\gamma : \mathrm{SP}$.

Further each $\mathrm{Arg}\,\gamma$ is κ-based, if κ is greater than the cardinality of each set occurring as a side condition or an arity in γ. A regular κ' with this property always exists (using classical logic).

Hence $I(\mathrm{Arg}\,\gamma)$ exists.

The elimination principle can be interpreted using that $I(\mathrm{Arg}\,\gamma)$ is the least fixed point.

Generic programming

One advantage of a uniform presentation of inductive definitions is that it explains what they all have in common.

Generic programming

One advantage of a uniform presentation of inductive definitions is that it explains what they all have in common.

Another is that we can now reason about inductive definitions internally, by reasoning about $\gamma : \mathsf{SP}$.

Generic programming

One advantage of a uniform presentation of inductive definitions is that it explains what they all have in common.

Another is that we can now reason about inductive definitions internally, by reasoning about $\gamma : \mathsf{SP}$.

Many programming languages have special facilities for writing generic programs that can be instantiated to concrete data types, e.g. `deriving Eq` in Haskell.

Generic programming

One advantage of a uniform presentation of inductive definitions is that it explains what they all have in common.

Another is that we can now reason about inductive definitions internally, by reasoning about $\gamma : \mathsf{SP}$.

Many programming languages have special facilities for writing generic programs that can be instantiated to concrete data types, e.g. `deriving Eq` in Haskell.

With a universe of inductive definitions, we have

$$\text{Generic programming} = \text{Programming}$$

Generic programming

One advantage of a uniform presentation of inductive definitions is that it explains what they all have in common.

Another is that we can now reason about inductive definitions internally, by reasoning about $\gamma : \mathsf{SP}$.

Many programming languages have special facilities for writing generic programs that can be instantiated to concrete data types, e.g. `deriving Eq` in Haskell.

With a universe of inductive definitions, we have

$$\text{Generic programming} = \text{Programming}$$

A more high-level language with data declarations etc can be elaborated to codes using the universe of inductive definitions.

Example: decidable equality

Let us implement `deriving Eq`.

Rather than just saying yes or no, let us also produce evidence that $\mathrm{Dec}\,(x =_A y) :\equiv (x =_A y) + (x \neq_A y)$.

Example: decidable equality

Let us implement `deriving Eq`.

Rather than just saying yes or no, let us also produce evidence that $\mathrm{Dec}\,(x =_A y) :\equiv (x =_A y) + (x \neq_A y)$.

Obviously side conditions need to have decidable equality, and we only allow finitary constructors.

Example: decidable equality

Let us implement `deriving Eq`.

Rather than just saying yes or no, let us also produce evidence that $\mathrm{Dec}\,(x =_A y) :\equiv (x =_A y) + (x \neq_A y)$.

Obviously side conditions need to have decidable equality, and we only allow finitary constructors.

We mutually define

$$\mathrm{eq} : (\gamma : \mathrm{SP}) \to (x : D_\gamma) \to (y : D_\gamma) \to \mathrm{Dec}\,(x = y)$$
$$\mathrm{eqArg} : (\gamma, \gamma' : \mathrm{SP}) \to (x : \mathrm{Arg}\,\gamma\,D_{\gamma'}) \to (y : \mathrm{Arg}\,\gamma\,D_{\gamma'}) \to \mathrm{Dec}\,(x = y)$$

emacs27@pl-fredri.cis.strath.ac.uk <2>

File Edit Options Buffers Tools Agda Help

```
open import Data.Product
open import Data.Product.Properties
open import Data.Unit
open import Relation.Nullary
open import Relation.Binary.PropositionalEquality

open import Axiom.UniquenessOfIdentityProofs

module decEq (U : Set)(T : U → Set)
             (dec : (A : U) → (x y : T A) → Dec (x ≡ y)) where

-- universe containing unit type only
T⊤ : ⊤ → Set
T⊤ _ = ⊤

-- Because of uniqueness for ⊤ and functions, we can prove funext for
-- functions out of ⊤
funext⊤ : {A : Set} → {f g : ⊤ → A} → f tt ≡ g tt → f ≡ g
funext⊤ {A} {f} {g} = lem {A} {f tt} {g tt}
  where
   lem : {A : Set} → {ftt gtt : A} → ftt ≡ gtt → (λ (x : ⊤) → ftt) ≡ (λ x → gtt)
   lem refl = refl

open import SP U T ⊤ T⊤

mutual

  eqArg : {s' : SP}(s : SP) → (x y : Arg s (D s')) → Dec (x ≡ y)
  eqArg done x y = yes refl
  eqArg (non-ind A s) (a , x) (a' , y) with dec A a a'
  eqArg (non-ind A s) (a , x) (a , y) | yes refl with eqArg (s a) x y
  eqArg (non-ind A s) (a , x) (a , y) | yes refl | yes x≡y =
    yes (cong (a ,_) x≡y)
  eqArg (non-ind A s) (a , x) (a , y) | yes refl | no ¬x≡y =
    no λ r → ¬x≡y (,-injectiveʳ-≡ (Decidable⇒UIP.≡-irrelevant (dec A)) r refl)
  eqArg (non-ind A s) (a , x) (a' , y)| no ¬p =
    no λ ax≡a'y → ¬p (cong proj₁ ax≡a'y)
  eqArg {s'} (ind A s) (f , x) (g , y) with eq s' (f _) (g _) | eqArg s x y
  ... | yes p | yes q = yes (cong₂ _,_ (funext⊤ p) q)
  ... | yes p | no q = no λ r → q (cong proj₂ r)
  ... | no p | q = no λ r → p (cong (λ h → h tt) (cong proj₁ r))

  eq : (s : SP) → (x y : D s) → Dec (x ≡ y)
  eq s (inn x) (inn y) with eqArg s x y
  ... | yes p = yes (cong inn p)
  ... | no ¬p = no (λ innx≡inny → ¬p (inn-inj innx≡inny))
```

-U:--- decEq.agda All L47 (Agda:Checked)

-U:%*- *All Done* All L1 (AgdaInfo)

Summary

Generic treatment of inductive definitions in type theory using a universe of data types.

Can be given set-theoretic semantics using iteration of monotone operators.

In type theory, "Generic programming" = "Programming".