

---

# Inductive-Recursive Definitions and Generic Programming

Anton Setzer  
Swansea University

1. What is Generic Programming?
2. Universes and Generic Programming
3. Inductive-Recursive Definitions

# Notations

---

- $\text{Set}$  = “small types”.



$$\underline{(x : A, y : B) \rightarrow C}$$

is type of functions,

- mapping  $x : A, y : B$  to  $C$ ,
  - where  $B$  might depend on  $x$ ,
  - $C$  might depend on  $x, y$ .
  - Sometimes written as  $\Pi x : A. \Pi y : B. C$ .
- $\underline{(x : A) \times B}$  = dependent product ( $B$  might depend on  $A$ ).
  - Sometimes written as  $\Sigma x : A. B$ .

# 1. What is Generic Programming?

---

- Two different notions:
  - In **object-oriented programming**: higher type polymorphism.
    - E.g. to form  $\text{List}(A)$  depending on  $A : \text{Set}$ .
  - In **functional programming**:
    - Definition of functions by induction on the buildup of types.
    - **Notion used here.**
  - Notion of **Generative Programming**
    - Idea of automatic generation of programs.
    - “Never write the same code twice”.

# 2. Universes

---

- Universes in type theory = general framework for generic programming.
- Universe given by
  - a **set of codes** for sets

$U : \text{Set}$

- a **decoding function**

$T : U \rightarrow \text{Set}$

# Examples

---

1.

$$U = \mathbb{N} , \quad T(n) := \mathbb{N}^n$$

2.

$$\begin{aligned} \text{data } U &= \widehat{\mathbb{N}} \mid (\widehat{\Rightarrow}) (a, b : U) \mid (\widehat{\times}) (a, b : U) , \\ T \widehat{\mathbb{N}} &= \mathbb{N} \\ T (a \widehat{\Rightarrow} b) &= T a \rightarrow T b \\ T (a \widehat{\times} b) &= T a \times T b \end{aligned}$$

# Universes and Generic Programm.

---

- Universes allow to define generic functions
  - Assume universes  $(U_0, T_0)$ ,  $(U_1, T_1)$  and  $A : \text{Set}$ .
  - A **generic function** can be given as a function

$$f : U_0 \rightarrow A$$

or

$$f : U_0 \rightarrow U_1$$

or

$$f : (u : U_0) \rightarrow T_0 u \rightarrow ((u' : U_1) \times T_1 u')$$

or

...

- If  $U_0$  is inductively defined,  $f$  can be defined by induction on  $U_0$ .

# Example Module Structure

---

$\text{ModStruct} = \text{data ms } (l : \text{List } ((\text{weight} : \mathbb{R}) \times \text{ModStruct}))$

$\text{Marks} : \text{ModStruct} \rightarrow \text{Set}$

$\text{Marks } (\text{ms } []) = \mathbb{R}$

$\text{Marks } (\text{ms } [\langle w_0, l_0 \rangle, \dots, \langle w_n, l_n \rangle])$   
 $= \text{Marks } l_0 \times \dots \times \text{Marks } l_n$

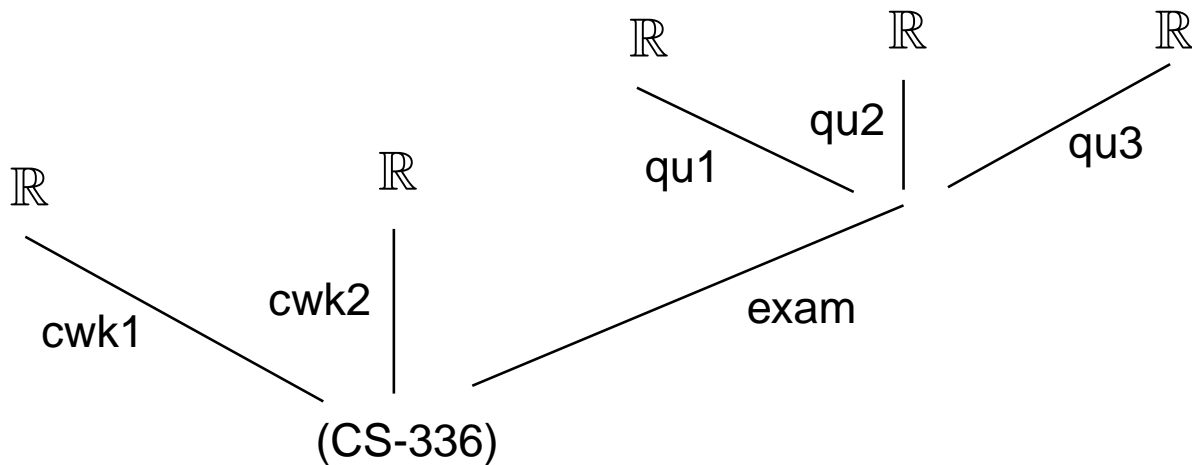
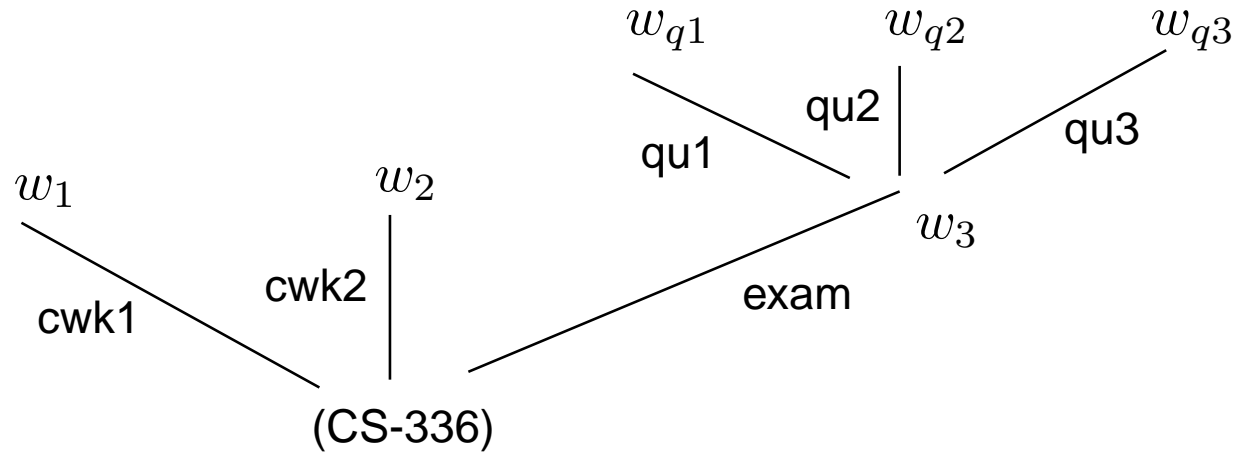
$\text{result} : (s : \text{ModStruct}, \text{Marks } s) \rightarrow \mathbb{R}$

$\text{result } (\text{ms } []) r = r$

$\text{result } (\text{ms } [\langle w_0, l_0 \rangle, \dots, \langle w_n, l_n \rangle]) \langle m_0, \dots, m_n \rangle$   
 $= \frac{\text{result } l_0 m_0 * w_0 + \dots + \text{result } l_n m_n * w_n}{w_0 + \dots + w_n}$

# Module Structure

---



# Example Module Structure

---

- In order to define a show function, one needs to modify

$$\text{ModStruct} = \text{data ms } (\textit{name} : \text{String}) \\ (\textit{l} : \text{List } ((\textit{weight} : \mathbb{R}) \times \text{ModStruct})) .$$

- $\Rightarrow$  we don't have “the generic data structure”  
instead there are lots of generic data structures

# Gen. Program. on Ind. Data Types

---

- We want to carry out generic programming on the set of **inductively defined data types**.
  - E.g. the **show** function, **equality** on data types, the **zipper** etc.
- Inductively defined sets are **initial algebras for strictly positive functors**.
- Use of **universes of operators**

$$U : \text{Set} \quad F : U \rightarrow \text{Set} \rightarrow \text{Set}$$

# Generic Haskell

---

- One version of **Generic Haskell** corresponds to the following universe:

(here  $U_0, T_0 : U_0 \rightarrow \text{Set}$  is a universe of basic sets):

$$\text{data } U = \text{id} \mid \text{K } (a : U_0) \mid U \hat{+} U \mid U \hat{\times} U \\ \mid \text{is\_of } (s : \text{String}) (u : U)$$
$$F_{\text{id}} = \lambda X. X$$
$$F_{\text{K } a} = \lambda X. T_0 a$$
$$F_{a \hat{+} b} = \lambda X. F_a X + F_b X$$
$$F_{(a \hat{\times} b)} = \lambda X. F_a X \times F_b X$$
$$F_{\text{is\_of } s u} = \lambda X. F_u X$$

- Then define for  $f : U$   
 $A_f = \text{data intro}_f (x : F_f A_f)$

# Generic Haskell

---

- Examples in generic Haskell (e.g. **zipper**) can now easily be transformed into generic functions in dependent type theory.
  - Amounts to defining generic functions on the universe  $\mathcal{U}$  of universe operators.

# Dependent Inductive Definitions

---

- **Dependent inductive definitions** can be defined (referring to a basic universe of sets  $U_0, T_0$ ) by defining

$$U : \text{Set} \quad F : U \rightarrow \text{Set} \rightarrow \text{Set}$$

with the following elements:

- $\iota : U, \quad F_\iota X = \{*\}$ 
  - Let  $f := \iota$ . Then

$$\text{intro}_f : \{*\} \rightarrow A_f$$

Corresponds to the constructor having essentially **no argument**.

# Dependent Inductive Definitions

---

•  $\sigma : (a : U_0, g : T_0 a \rightarrow U) \rightarrow U$

$F_{\sigma a g} X = (x : T_0 a) \times F_{g x} X$

• Let  $f := \sigma a g$ . Then

$\text{intro}_f : ((x : T_0 a) \times (F_{g x} A_f)) \rightarrow A_f$

Corresponds to the constructor having **one non-inductive argument**, and **later arguments depending on it**.

# Dependent Inductive Definitions

---

- $\delta : (a : U_0, u : U) \rightarrow U$   
 $F_{\delta a u} X = (T_0 a \rightarrow X) \times F_u X$

- Let  $f := \delta a u$ . Then

$$\text{intro}_f : ((T_0 a \rightarrow A_f) \times F_u A_f) \rightarrow A_f$$

Corresponds to the constructor having one **inductive argument**, and **later arguments given by  $u$** .

- Variants for this have been used by Benke/Dybjer/Jansson e.g. for defining decidable equality on finitary data types and showing in type theory that this is an equivalence relation.

# 3. Inductive-Recursive Definitions

---

- General theory of universes

$$U : \text{Set} \quad T : U \rightarrow D$$

for some type  $D$ .

- Example Universe closed under  $\times$  has constructor

$$\begin{aligned} (\widehat{\times}) &: (a : U, b : T a \rightarrow U) \rightarrow U \\ T ((\widehat{\times}) a b) &= (x : T a) \times T (b x) \end{aligned}$$

- Amounts to having **initial algebras for endo functors**

$$F : \text{Fam}_D \rightarrow \text{Fam}_D$$

where

$$\text{Fam}_D := (X : \text{Set}) \times (X \rightarrow D)$$

# Generic Program. on Ind.-rec. Def.

---

- Now one can define a universe of endo functors

$$\text{OP}_D : \text{Type} \quad \text{F}_D : \text{OP}_D \rightarrow \text{Fam}_D \rightarrow \text{Fam}_D$$

and define **inductively-recursively** the universes given by  $\text{OP}_D$

$$\text{U}_D : \text{OP}_D \rightarrow \text{Set}$$

$$\text{T}_D : (\gamma : \text{OP}_D, u : \text{U}_{D,\gamma}) \rightarrow D$$

s.t.  $\text{OP}_D$  contains codes for all ind-rec. definitions.

- Type theory with relatively few rules, which is highly expressive.
- Allows generic programming on inductive-recursive definitions.

# Conclusion

---

- Dependent type theory and universes provide an excellent **framework for generic programming**.
  - **Not much difference to ordinary programming**.
- There is **not only one generic data structure**, but there are many which can be custom built.
- Closed inductive-recursive definitions provide a **highly expressive dependent type theory**.
- Richness of inductive-recursive definitions is still to be discovered by general computer science.
- Literature: Articles with **P. Dybjer** on **inductive-recursive definitions** available from  
**<http://www.cs.swan.ac.uk/~setzer>**