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

- $\text{Set} = “\text{small types”}.$

\[(x : A, y : B) \rightarrow C\]

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 $\prod x : A.\prod y : B.C.$.

\[(x : A) \times B\] = 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”.

Anton Setzer: Inductive Recursive Definitions
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

\[ T : U \to \text{Set} \]
Examples

1. \[ U = \mathbb{N} \ , \quad T(n) := \mathbb{N}^n \]

2. 

\[
\begin{align*}
\text{data } U &= \hat{\mathbb{N}} \mid (\leftrightarrow) (a, b : U) \mid (\times) (a, b : U) , \\
T \hat{\mathbb{N}} &= \mathbb{N} \\
T (a \leftrightarrow b) &= T a \to T b \\
T (a \times b) &= T a \times T b
\end{align*}
\]
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

\[
\begin{align*}
  f &: \ U_0 \rightarrow A \\
  \text{or} \\
  f &: \ U_0 \rightarrow U_1 \\
  \text{or} \\
  f &: \ (u : U_0) \rightarrow T_0 \ u \rightarrow ((u' : U_1) \times T_1 \ u') \\
  \text{or} \\
  \ldots
\end{align*}
\]

- If \(U_0\) is inductively defined, \(f\) can be defined by induction on \(U_0\).
Example Module Structure

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

\[ \text{Marks : ModStruct} \rightarrow \text{Set} \]
\[ \text{Marks (ms [ ])} = \mathbb{R} \]
\[ \text{Marks (ms [ } \langle w_0, l_0 \rangle , \ldots , \langle w_n , l_n \rangle \} ) \]
\[ = \text{Marks } l_0 \times \cdots \times \text{Marks } l_n \]

\[ \text{result : (} s : \text{ModStruct}, \text{Marks } s ) \rightarrow \mathbb{R} \]
\[ \text{result (ms [ ])} r = r \]
\[ \text{result (ms [ } \langle w_0 , l_0 \rangle , \ldots , \langle w_n , l_n \rangle \} ) \langle m_0 , \ldots , m_n \rangle \]
\[ = \frac{\text{result } l_0 m_0 * w_0 + \cdots + \text{result } l_n m_n * w_n}{w_0 + \cdots w_n} \]
Module Structure

(CS-336)

$w_1$

$cwk1$

$cwk2$

$w_2$

$w_q1$

$qu1$

$w_3$

$w_q2$

$qu2$

$w_q3$

$qu3$

$RR$

$RR$

$cwk1$

$cwk2$

$(CS-336)$

$RR$
Example Module Structure

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

\[
\text{ModStruct} = \text{data ms } (\text{name : String}) \\
\hspace{1cm} (l : \text{List } ((\text{weight : } \mathbb{R}) \times \text{ModStruct})) .
\]

⇒ we don’t have “the generic data structure”
instead there are lots of generic data structures
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):

```
data U = id | K (a : U_0) | U \dagger U | U \times U
       | is_of (s : String) (u : U)

F_{id} = \lambda X. X
F_{K a} = \lambda X. T_0 a
F_{a \dagger b} = \lambda X. F_a X + F_b X
F_{(a \times b)} = \lambda X. F_a X \times F_b X
F_{is\_of s u} = \lambda X. F_u X
```

Then define for $f : U$

$$A_f = \text{data intro}_f (x : F_f A_f)$$
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 $\mathbb{U}$ 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

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

with the following elements:

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

$$\text{intro}_f : \{\ast\} \to A_f$$

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
  
  \[ U : \text{Set} \quad T : U \to D \]

  for some type \( D \).

- Example Universe closed under \( \times \) has constructor
  
  \[
  (\times) : (a : U, b : T a \to U) \to U \\
  T ((\times) a b) = (x : T a) \times T (b x)
  \]

- Amounts to having **initial algebras for endo functors**
  
  \[ F : \text{Fam}_D \to \text{Fam}_D \]

  where
  
  \[
  \text{Fam}_D := (X : \text{Set}) \times (X \to D)
  \]
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 \textbf{inductive-recursively} the universes given by \text{OP}_D

\[ U_D : \text{OP}_D \rightarrow \text{Set} \]
\[ T_D : (\gamma : \text{OP}_D, u : U_D, \gamma) \rightarrow D \]

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 provides 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