Analysis and synthesis of inductive families

Hsiang-Shang Ko

9 October 2013

Contents

Chapter 1

Introduction

Chapter 2

From intuitionistic type theory to dependently typed programming

We start with an introduction to intuitionistic type theory [Martin-Löf, 1984] and dependently typed programming [Altenkirch et al., 2005; McBride, 2004] using the Agda language [Norell, 2007, 2009; Bove and Dybjer, 2009]. Intuitionistic type theory was developed by Martin-Löf to serve as a foundation of intuitionistic mathematics like Bishop's renowned work on constructive analysis [Bishop and Bridges, 1985]. While originated from intuitionistic type theory, dependently typed programming is more concerned with mechanisation and practicalities, and is influenced by the program construction movement. It has thus departed from the mathematical traditions considerably, and deviations can be found from syntactic presentations to the underlying philosophy.

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
```

Chapter 3

Refinements and ornaments

"datatypes" for inductive families

3.1 Datatype descriptions

$$\mathsf{Desc} : (I : \mathsf{Set}) \to \mathsf{Set}_1$$

$$\mu : \{I : \mathsf{Set}\} \to \mathsf{Desc}\ I \to (I \to \mathsf{Set})$$

$$\begin{aligned}
&\textbf{data}\ \mathsf{RDesc}\ (I : \mathsf{Set}) : \mathsf{Set}_1\ \textbf{where}\\
&\quad \mathsf{v} : (is : \mathsf{List}\ I) \to \mathsf{RDesc}\ I\\
&\quad \sigma : (S : \mathsf{Set})\ (D : S \to \mathsf{RDesc}\ I) \to \mathsf{RDesc}\ I\\
&[\![_]\!] : \{I : \mathsf{Set}\} \to \mathsf{RDesc}\ I \to (I \to \mathsf{Set}) \to \mathsf{Set}\\
&[\![\ \mathsf{v}\ is\ \ ]\!]\ X = \mathbb{M}\ is\ X\\
&[\![\ \sigma\ S\ D\ ]\!]\ X = \Sigma[\,s : S\,]\ [\![\ D\ s\ ]\!]\ X
\end{aligned}$$

$$\begin{aligned}
&\mathbb{M} : \{I : \mathsf{Set}\} \to \mathsf{List}\ I \to (I \to \mathsf{Set}) \to \mathsf{Set}\\
&\mathbb{M}\ [\,]\ \ \ \ X = \top\\
&\mathbb{M}\ (i :: is)\ X = X\ i \times \mathbb{M}\ is\ X
\end{aligned}$$

3.2 Ornaments

Evolutionary remark. We define ornaments as relations between descriptions (indexed by an erasure function), whereas McBride's original ornaments [2011] are rebranded as ornamental descriptions. One obvious advantage of relational ornaments is that they can arise between *existing* descriptions, whereas ornamental descriptions always produce (definitionally) new descriptions at the more informative end. This also means that there can be multiple ornaments between a pair of descriptions. For example, consider the datatype

$$\textbf{indexfirst data}\ \mathsf{Square}\ (A : \mathsf{Set}) : \mathsf{Set}\ \textbf{where}$$
$$\quad \mathsf{Square}\ A \ni \_,\_\ (x : A)\ (y : A)$$

Between the description of Square A and itself, we have the identity ornament

$$\sigma[x : A]\ \sigma[y : A] \vee \mathsf{tt}$$

and the ornament

$$\Delta[x : A]\ \Delta[y : A]\ \nabla[y]\ \nabla[x] \vee \mathsf{tt}$$

whose forgetful function swaps the fields x and y. The other advantage of relational ornaments is that they allow new datatypes to arise at the less informative end. For example, *coproduct of signatures* as used in, e.g., data types à la carte [Swierstra, 2008], can be implemented naturally with relational ornaments but not with ornamental descriptions. In more detail: Consider (a simplistic variation of) *tagged descriptions* [Chapman et al., 2010], which are descriptions that, for any index request, always respond with a constructor field first. A tagged description with index set $I : \mathsf{Set}$ thus consists of a family of types $C : I \to \mathsf{Set}$, where each $C\ i$ is the set of constructor tags for the index request $i : I$, and a family of subsequent response descriptions for each constructor tag.

$$\mathsf{TDesc} : \mathsf{Set} \to \mathsf{Set}_1$$
$$\mathsf{TDesc}\ I = \Sigma[C : I \to \mathsf{Set}]\ ((i : I) \to C\ i \to \mathsf{RDesc}\ I)$$

Tagged descriptions are decoded to ordinary descriptions by

$$\lfloor \_ \rfloor_T : \{I : \mathsf{Set}\} \to \mathsf{TDesc}\ I \to \mathsf{Desc}\ I$$
$$\lfloor C, D \rfloor_T\ i = \sigma\ (C\ i)\ (D\ i)$$

We can then define binary coproduct of tagged descriptions, which sums up the corresponding constructor fields, as follows:

$$\_\oplus\_ : \{I : \mathsf{Set}\} \to \mathsf{TDesc}\ I \to \mathsf{TDesc}\ I \to \mathsf{TDesc}\ I$$
$$(C, D) \oplus (C', D') = (\lambda\, i \mapsto C\ i + C'\ i), (\lambda\, i \mapsto D\ i \triangledown D'\ i)$$

Now given two tagged descriptions $tD = (C, D)$ and $tD' = (C', D')$ of type $\mathsf{TDesc}\ I$, there are two ornaments from $\lfloor tD \oplus tD' \rfloor_T$ to $\lfloor tD \rfloor_T$ and $\lfloor tD' \rfloor_T$

$$inlOrn : \mathsf{Orn}\ id\ \lfloor tD \oplus tD' \rfloor_T\ \lfloor tD \rfloor_T$$
$$inlOrn\ i = \Delta[c : C\ i]\ \nabla[\mathsf{inl}\ c]\ idOrn\ (D\ i\ c)$$
$$inrOrn : \mathsf{Orn}\ id\ \lfloor tD \oplus tD' \rfloor_T\ \lfloor tD' \rfloor_T$$
$$inrOrn\ i = \Delta[c' : C'\ i]\ \nabla[\mathsf{inr}\ c']\ idOrn\ (D'\ i\ c')$$

whose forgetful functions perform suitable injection of constructor tags. Note that the synthesised new description $\lfloor tD \oplus tD' \rfloor_T$ is at the less informative end of *inlOrn* and *inrOrn*. (*End of evolutionary remark.*)

Examples?

Chapter 4

Categorical organisation of the ornament–refinement framework

Chapter 5

Relational algebraic ornaments

The three datatypes Nat, $\mathsf{List}\ A$, and $\mathit{Vec}\ A$ are evidently related: a list is a natural number whose cons nodes are decorated with elements of A, and a vector is a list enriched with length information. Such relationship can be seen by "overlaying" one datatype declaration on the other: for example, the declaration of $\mathsf{List}\ A$ differs from that of Nat only in an extra field $(a\ :\ A)$ in the cons constructor, and the declaration of $\mathit{Vec}\ A$ differs from that of $\mathsf{List}\ A$ in that (i) the index set is changed from $\top$ to Nat, (ii) the cons constructor has two extra fields, and (iii) the index of the recursive position is specified to be m. Such differences between datatype declarations are encoded as *ornaments*. Whenever there is an ornament between two datatypes, there is a forgetful function from the more informative datatype to the other, erasing information according to the ornament's specification of datatype differences. For example, we have a forgetful function from lists to natural numbers that discards elements associated with cons nodes — i.e., it computes the length of a list — and another one from vectors to lists which removes all length information from a vector and returns the underlying list.

Ornaments constitute the second underlying universe:

$$\mathsf{Orn}\ :\ \{I\ J\ :\ \mathsf{Set}\}\ (e\ :\ J \to I)\ (D\ :\ \mathsf{Desc}\ I)\ (E\ :\ \mathsf{Desc}\ J) \to \mathsf{Set}_1$$

An ornament $O\ :\ \mathsf{Orn}\ e\ D\ E$ specifies the difference between the more informative description E and the basic description D, and is parametrised by an "index erasure" function e from the index set of E to that of D. The ornament gives rise to a forgetful function

$$\mathit{forget}\ O\ :\ \mu\ E\ \rightrightarrows\ (\mu\ D \circ e)$$

For example, there are families of ornaments

$$\mathit{NatD} - \mathit{ListD}\ :\ (A\ :\ \mathsf{Set}) \to \mathsf{Orn}\ !\ \mathit{NatD}\ (\mathit{ListD}\ A)$$

and

$$\mathit{ListD} - \mathit{VecD}\ :\ (A\ :\ \mathsf{Set}) \to \mathsf{Orn}\ !\ (\mathit{ListD}\ A)\ (\mathit{VecD}\ A)$$

(where $!\ =\ \mathit{const}\ \mathsf{tt}$) that encode the differences between the list-like datatypes. The function

$$\mathit{forget}\ (\mathit{NatD} - \mathit{ListD}\ A)\ \{\mathsf{tt}\}\ :\ \mathsf{List}\ A \to \mathsf{Nat}$$

computes the length of a list, and the function

$$\mathit{forget}\ (\mathit{ListD} - \mathit{VecD}\ A)\ :\ \forall\ \{n\} \to \mathit{Vec}\ A\ n \to \mathsf{List}\ A$$

computes the underlying list of a vector.

Ornamental descriptions. Ornaments arise between existing datatype descriptions. The typical scenario of using ornaments, however, is first modifying a base description into a more informative one and then specifying an ornament between the two descriptions. *Ornamental descriptions* are introduced to combine the two steps into one:

$$\mathit{OrnDesc}\ :\ \{I\ :\ \mathsf{Set}\}\ (J\ :\ \mathsf{Set})\ (e\ :\ J \to I)\ (D\ :\ \mathsf{Desc}\ I) \to \mathsf{Set}_1$$

An ornamental description

$$\mathit{OD}\ :\ \mathit{OrnDesc}\ J\ e\ D$$

is like a new description of type Desc J, but is written relative to a base description D such that not only can we extract the new description

$$\lfloor \mathit{OD} \rfloor\ :\ \mathsf{Desc}\ J$$

but we can also extract an ornament from the base description D to the new description

$$\lceil\ \mathit{OD}\ \rceil\ :\ \mathsf{Orn}\ e\ D\ \lfloor \mathit{OD} \rfloor$$

An ornamental description is a convenient way to specify a new datatype that has an ornamental relationship with an existing one; it might be thought of as simultaneously denoting the new description and the ornament — the floor and ceiling brackets $\lfloor \_ \rfloor$ and $\lceil \_ \rceil$ are added to resolve ambiguity.

Example. Let $\_ \leqslant A\_ : A \to A \to \mathsf{Set}$ be an ordering on A and declare a datatype of ordered lists (parametrised by A and $\_ \leqslant A\_$) indexed by a lower bound under this ordering:

```
indexfirst data OrdList A _⩽A_ : A → Set where
  OrdList A _⩽A_ b
    accepts nil
    or      cons (a : A) (leq : b ⩽ A a) (as : OrdList A _⩽A_ a)
```

This datatype can be thought of as being decoded from an ornamental description

$$\mathit{OrdListOD}\ A\ \_ \leqslant A\_\ :\ \mathit{OrnDesc}\ A\ !\ (\mathit{ListD}\ A)$$

which inserts the field *leq* and refines the index of the recursive position to a. That is, the underlying description for *OrdList* is

$$\lfloor \mathit{OrdListOD}\ A\ \_ \leqslant A\_ \rfloor\ :\ \mathsf{Desc}\ A$$

(so $\mathit{OrdList}\ A\ \_ \leqslant A\_\ b$ desugars to $\mu\ \lfloor \mathit{OrdListOD}\ A\ \_ \leqslant A\_ \rfloor\ b$), and

$$\lceil\ \mathit{OrdListOD}\ A\ \_ \leqslant A\_\ \rceil\ :\ \mathsf{Orn}\ !\ (\mathit{ListD}\ A)\ \lfloor \mathit{OrdListOD}\ A\ \_ \leqslant A\_ \rfloor$$

is the ornament from lists to ordered lists.

Chapter 6

Categorical equivalence of ornaments and relational algebras

Chapter 7

Conclusion

7.1 Future work

- functor-level abstraction

Bibliography

Thorsten Altenkirch, Conor McBride, and James McKinna [2005]. Why dependent types matter. Available at `http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf`.

Errett Bishop and Douglas Bridges [1985]. *Constructive Analysis*. Springer-Verlag.

Ana Bove and Peter Dybjer [2009]. Dependent types at work. In *Language Engineering and Rigorous Software Development*, volume 5520 of *Lecture Notes in Computer Science*, pages 57–99. Springer-Verlag.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris [2010]. The gentle art of levitation. In *International Conference on Functional Programming*, ICFP'10, pages 3–14. ACM.

Per Martin-Löf [1984]. *Intuitionistic Type Theory*. Bibliopolis, Napoli.

Conor McBride [2004]. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170.

Conor McBride [2011]. Ornamental algebras, algebraic ornaments. To appear in *Journal of Functional Programming*.

Ulf Norell [2007]. *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology.

Ulf Norell [2009]. Dependently typed programming in Agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer-Verlag.

Wouter Swierstra [2008]. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436. doi:10.1017/S0956796808006758.