Universes for Data

Peter Morris

University of Nottingham

November 12, 2009

Outline

Roadmap

Curry-Howard and Dependently Typed Programming

- Dependently Typed Programming is based on the idea that Types are Propositions, and Programs are Proofs. This is the Curry-Howard Isomorphism.
- First we identify the type of propositions with Set.
- The implication $A \implies B$ is a function $A \to B$
- The conjunction $A \wedge B$ is a Cartesian-product $A \times B$
- The disjunction $A \vee B$ is a disjoint union $A + B$
- So we can interpret Propositional Logic as a simply typed lambda calculus.
- But what about Predicate logic?

Curry-Howard and Predicate Logic

- A predicate on A is a function $P : A \to$ Set
- How do we interpret the proposition $\forall a : A.P\, a$?
- As a dependent function space (a : A) → P a.
- The type of the output of such a function, varies depending on the input.
- What the proposition $\exists a : A.Pa$?
- We'll come back to that...

Indexed Families

The datatypes of dependently typed languages can also depend on data:

Natural Numbers

data Nat : Set **where**
zero : Nat
succ : (n : Nat) → Nat

Lists

data List (A : Set) : Set **where**
ϵ : List A
_::_ : (a : A) (as : List A) → List A

Finer Program Control

- We can use these indicies to prevent programs from going wrong

Safe hd

$$\begin{aligned} &\mathsf{hd} \;:\; \forall \{n\}\,\{A\} \to \mathsf{Vec}\ A\ (\mathsf{succ}\ n) \to A \\ &\mathsf{hd}\ (a :: as) \;=\; a \end{aligned}$$

- Compare with the version for lists:

Maybe hd

$$\begin{aligned} &\mathsf{maybehd} \;:\; \forall \{A\} \to \mathsf{List}\ A \to \mathsf{Maybe}\ A \\ &\mathsf{maybehd}\ \epsilon \;=\; \mathsf{no} \\ &\mathsf{maybehd}\ (a :: as) \;=\; \mathsf{yes}\ a \end{aligned}$$

Finite Sets

- We can define a set Fin n which has exactly n elements:

Finite Sets

```
data Fin  : Nat → Set where
  zero : ∀ {n}          → Fin (succ n)
  succ : ∀ {n} → Fin n → Fin (succ n)
```

- Which can help define a type of well scoped lambda terms:

Scoped Lambda-Terms

```
data Lam  : Nat → Set where
  var : ∀ {n} → Fin n              → Lam n
  app : ∀ {n} → Lam n → Lam n      → Lam n
  abs : ∀ {n} → Lam (succ n)       → Lam n
```

Existential quantification and equality

- Using these indexed families, we can return to the question of interpreting existentials - as Sigma-types:

Sigma Types

data Σ (A : Set) (P : A → Set) : Set **where**
  _,_ : (x : A) (y : P x) → Σ A P

- so $\exists a : A.P\,a$ is interpreted as Σ A \a → P a
- We can also define predicates and relations inductively, for instance equality:

Sigma Types

data _≡_ {A : Set} (a : A) : A → Set **where**
  refl : a ≡ a

Schemas

- What is the status of these Datatypes with respect to the Type Theory of the programming language?
- We have to be careful of what definitions we allow...
- With languages like Agda, and Epigram an external piece of code, a *schema checker*, looks to see if each definition is OK with a syntactic check.
- If it is the TT is extended with the introduction, computation and equality rules for the data type.
- This approach, however, brings about problems for reasoning about the language, we need an external framework to prove the schema checker correct.
- It also precludes any attempt to interpret the language in itself, Agda in Adga.

Universes

- Informally universe is a collection of types (sets).
- Russell's solution to the paradoxes of Set-theory was to introduce a predicative hierarchy of universes:

Russell's Universe Hierarchy

$$\mathsf{Set}_0 : \mathsf{Set}_1 : \mathsf{Set}_2 : \ldots : \mathsf{Set}_i : \mathsf{Set}_{i+1} : \ldots\ldots$$

- Or alternatively:

Tarski's Universe Hierarchy

$$\begin{array}{rcl} \mathsf{U}_i & : & \mathsf{Set} \\ \mathsf{El}_i & : & \mathsf{U}_i \rightarrow \mathsf{Set} \\ \mathsf{u}_i & : & \mathsf{U}_{i+1} \\ \text{s.t. } \mathsf{El}_{i+1}\ \mathsf{u}_i \equiv \mathsf{U}_i \end{array}$$

Universes for Data

- We can use Tarski style universes to capture other interesting collections of types, in general we'll need:

Tarski's Universes

$$\begin{array}{rcl} \mathsf{U} & : & \mathsf{Set} \\ \mathsf{El} & : & \mathsf{U} \rightarrow \mathsf{Set} \end{array}$$

- If we can capture a universe of inductive families in our language, then we can do without external schemas.
- With such a universe, creating a datatype would no longer extend the logic, making it easier to reason about the system itself.

Roadmap

The Syntax of Inductive Types

- Lets start by simply encoding the syntax of data definitions:

A syntax

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Desc}\ :\ \mathsf{Set}\ \textbf{where}\\
&\quad \mathsf{done}\ :\ \mathsf{Desc}\\
&\quad \mathsf{arg}\ :\ (\mathsf{A}\ :\ \mathsf{Set}) \to (\phi\ :\ \mathsf{A} \to \mathsf{Desc}) \to \mathsf{Desc}\\
&\quad \mathsf{ind}\ :\ (\mathsf{H}\ :\ \mathsf{Set}) \to (\phi\ :\ \mathsf{Desc}) \to \mathsf{Desc}
\end{aligned}
$$

- Every data *description* gives rise to a functor:

Interpreting the syntax

$$
\begin{aligned}
&[\![\_]\!]\ :\ \mathsf{Desc} \to \mathsf{Set} \to \mathsf{Set}\\
&[\![\ \mathsf{done}\ ]\!]\quad \mathsf{D}\ =\ \mathbf{1}\\
&[\![\ \mathsf{arg}\ \mathsf{A}\ \phi\ ]\!]\ \mathsf{D}\ =\ \Sigma\ \mathsf{A}\ \backslash \mathsf{a} \to [\![\ \phi\ \mathsf{a}\ ]\!]\ \mathsf{D}\\
&[\![\ \mathsf{ind}\ \mathsf{H}\ \phi\ ]\!]\ \mathsf{D}\ =\ \Sigma\ (\mathsf{H} \to \mathsf{D})\ \backslash \mathsf{h} \to [\![\ \phi\ ]\!]\ \mathsf{D}
\end{aligned}
$$

The Syntax of Inductive Types (2)

- The initial algebras of these functors are our data types:

Initial Algebras

$$\textbf{data}\ \mu\ (\phi\ :\ \mathsf{Desc})\ :\ \mathsf{Set}\ \textbf{where}$$
$$\quad \mathsf{intro}\ :\ [\![\,\phi\,]\!]\ (\mu\ \phi) \rightarrow \mu\ \phi$$

- By adding this as a rule to our theory we encode introduction rules for all inductive types in one go.

The Syntax of Inductive Types (3)

Example

Lists

$$\begin{aligned}
&\mathsf{ListC} : \mathsf{Set} \to \mathsf{Desc}\\
&\mathsf{ListC}\ \mathsf{A} = \mathsf{arg}\ [\mathsf{cnil}\ \mathsf{ccons}]\ \backslash \mathsf{x} \to \mathsf{case}\ \mathsf{x}\ \mathsf{of}\\
&\quad \mathsf{cnil} \to \mathsf{done}\\
&\quad \mathsf{ccons} \to \mathsf{arg}\ \mathsf{A}\ \backslash_ \to \mathsf{ind}\ \mathbf{1}\ \mathsf{done}\\
&\mathsf{nil} : \{\mathsf{A} : \mathsf{Set}\} \to \mu\ (\mathsf{ListC}\ \mathsf{A})\\
&\mathsf{nil} : \mathsf{intro}\ (\mathsf{cnil}, _)\\
&\mathsf{cons} : \{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mu\ (\mathsf{ListC}\ \mathsf{A}) \to \mu\ (\mathsf{ListC}\ \mathsf{A})\\
&\mathsf{cons}\ \mathsf{a}\ \mathsf{as} = \mathsf{intro}\ (\mathsf{ccons}, (\mathsf{a}, \mathsf{as}, _))
\end{aligned}$$

Elimination

Induction

$$
\begin{aligned}
&\Box \;:\; (\phi : \mathsf{Desc})\ (\mathsf{D} : \mathsf{Set})\ (\mathsf{P} : \mathsf{D} \to \mathsf{Set})\ (\mathsf{v} : [\![\, \phi \,]\!]\ \mathsf{D}) \to \mathsf{Set}\\
&\Box\ \mathsf{done}\qquad \mathsf{D\ P\ v} \quad = \mathbf{1}\\
&\Box\ (\mathsf{arg\ A}\ \phi)\ \mathsf{D\ P}\ (\mathsf{a}, \mathsf{b}) \ = \Box\ (\phi\ \mathsf{a})\ \mathsf{D\ P\ b}\\
&\Box\ (\mathsf{ind\ H}\ \phi)\ \mathsf{D\ P}\ (\mathsf{a}, \mathsf{b}) \ = \Sigma\ ((\mathsf{h} : \mathsf{H}) \to \mathsf{P}\ (\mathsf{a\ h}))\ \backslash\_ \to \Box\ \phi\ \mathsf{D\ P\ b}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{map}\Box \;:\; (\phi : \mathsf{Desc})\ (\mathsf{D} : \mathsf{Set})\ (\mathsf{P} : \mathsf{D} \to \mathsf{Set})\ (\mathsf{p} : (\mathsf{d} : \mathsf{D}) \to \mathsf{P\ d})\\
&\qquad\qquad (\mathsf{v} : [\![\, \phi \,]\!]\ \mathsf{D}) \to \Box\ \phi\ \mathsf{D\ P\ v}\\
&\mathsf{map}\Box\ \mathsf{done}\qquad \mathsf{D\ P\ p\ v} \quad = \_\\
&\mathsf{map}\Box\ (\mathsf{arg\ A}\ \phi)\ \mathsf{D\ P\ p}\ (\mathsf{a}, \mathsf{b}) \ = \mathsf{map}\Box\ (\phi\ \mathsf{a})\ \mathsf{D\ P\ p\ b}\\
&\mathsf{map}\Box\ (\mathsf{ind\ H}\ \phi)\ \mathsf{D\ P\ p}\ (\mathsf{a}, \mathsf{b}) \ = (\backslash\mathsf{h} \to \mathsf{p}\ (\mathsf{a\ h})), \mathsf{map}\Box\ \phi\ \mathsf{D\ P\ p\ b}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{elim} \;:\; (\phi : \mathsf{Desc})\ (\mathsf{P} : \mu\ \phi \to \mathsf{Set})\\
&\qquad (\mathsf{p} : (\mathsf{x} : [\![\, \phi \,]\!]\ (\mu\ \phi)) \to \Box\ \phi\ (\mu\ \phi)\ \mathsf{P\ x} \to \mathsf{P}\ (\mathsf{intro\ x}))\\
&\quad (\mathsf{v} : \mu\ \phi) \to \mathsf{P\ v}\\
&\mathsf{elim}\ \phi\ \mathsf{P\ p}\ (\mathsf{intro\ v}) \ = \mathsf{p\ v}\ (\mathsf{map}\Box\ \phi\ (\mu\ \phi)\ \mathsf{P}\ (\mathsf{elim}\ \phi\ \mathsf{P\ p})\ \mathsf{v})
\end{aligned}
$$

The Syntax of Inductive Families

- We can extend our syntax to include the necessary indexing information:

A syntax

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Desc}\ (\mathsf{I} : \mathsf{Set}) : \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{done} : \mathsf{I} \rightarrow \mathsf{Desc}\ \mathsf{I} \\
\quad \mathsf{arg} : (\mathsf{A} : \mathsf{Set}) \rightarrow (\phi : \mathsf{A} \rightarrow \mathsf{Desc}\ \mathsf{I}) \rightarrow \mathsf{Desc}\ \mathsf{I} \\
\quad \mathsf{ind} : (\mathsf{H} : \mathsf{Set}) \rightarrow (\mathsf{is} : \mathsf{H} \rightarrow \mathsf{I}) \rightarrow (\phi : \mathsf{Desc}\ \mathsf{I}) \rightarrow \mathsf{Desc}\ \mathsf{I}
\end{array}
$$

- Every description gives rise to an *I-indexed functor*:

Interpreting the syntax

$$
\begin{array}{lcl}
[\![\_]\!] : \{\mathsf{I} : \mathsf{Set}\} \rightarrow \mathsf{Desc}\ \mathsf{I} \rightarrow (\mathsf{I} \rightarrow \mathsf{Set}) \rightarrow (\mathsf{I} \rightarrow \mathsf{Set}) \\
[\![\ \mathsf{done}\ \mathsf{j}\ ]\!]\ \mathsf{D}\ \mathsf{i} & = & \mathsf{i} \equiv \mathsf{j} \\
[\![\ \mathsf{arg}\ \mathsf{A}\ \phi\ ]\!]\ \mathsf{D}\ \mathsf{i} & = & \Sigma\ \mathsf{A}\ \backslash \mathsf{a} \rightarrow [\![\ \phi\ \mathsf{a}\ ]\!]\ \mathsf{D}\ \mathsf{i} \\
[\![\ \mathsf{ind}\ \mathsf{H}\ \mathsf{is}\ \phi\ ]\!]\ \mathsf{D}\ \mathsf{i} & = & \Sigma\ ((\mathsf{h} : \mathsf{H}) \rightarrow \mathsf{D}\ (\mathsf{is}\ \mathsf{h}))\ \backslash \mathsf{h} \rightarrow [\![\ \phi\ ]\!]\ \mathsf{D}\ \mathsf{i}
\end{array}
$$

The Syntax of Inductive Families (2)

- The initial algebras of these functors are our data types:

Initial Algebras

$$\textbf{data}\ \mu\ \{\mathsf{I} : \mathsf{Set}\}\ (\phi : \mathsf{Desc\ I}) : \mathsf{I} \to \mathsf{Set}\ \textbf{where}$$
$$\quad \mathsf{intro} : \{\mathsf{i} : \mathsf{I}\} \to [\![\ \phi\ ]\!]\ (\mu\ \phi)\ \mathsf{i} \to \mu\ \phi\ \mathsf{i}$$

- By adding this as a rule to our TT we encode introduction rules for all inductive families in one go.

The Syntax of Inductive Families (3)

Example

Vectors

```
VecC : Set → Desc Nat
VecC A  =  arg [cnil ccons] \x → case x of
   cnil   → done zero
   ccons → arg Nat \n →
              arg A \_ →
              ind 1 (\_ → n)
              done (succ n)
```

$$\mathsf{nil} \;:\; \{\mathsf{A} \;:\; \mathsf{Set}\} \to \mu\;(\mathsf{VecC}\;\mathsf{A})\;\mathsf{zero}$$
$$\mathsf{nil} \;:\; \mathsf{intro}\;(\mathsf{cnil}, \mathsf{refl})$$

$$\mathsf{cons} \;:\; \forall\,\{\mathsf{n}\;\mathsf{A}\} \to \mathsf{A} \to \mu\;(\mathsf{VecC}\;\mathsf{A})\;\mathsf{n} \to \mu\;(\mathsf{VecC}\;\mathsf{A})\;(\mathsf{succ}\;\mathsf{n})$$
$$\mathsf{cons}\;\{\mathsf{n}\}\;\mathsf{a}\;\mathsf{as} \;=\; \mathsf{intro}\;(\mathsf{ccons}, (\mathsf{n}, \mathsf{a}, \mathsf{as}, \mathsf{refl}))$$

Elimination

Induction

$$
\begin{array}{l}
\Box : \{I : Set\}\ (\phi : Desc\ I)\ (D : I \to Set)\ (P : \{i : I\} \to D\ i \to Set) \\
\quad \{i : I\} \to (v : [\![\ \phi\ ]\!]\ D\ i) \to Set \\
\Box\ (done\ i)\ \ D\ P\ refl = \mathbf{1} \\
\Box\ (arg\ A\ \phi)\ \ D\ P\ (a, b) = \Box\ (\phi\ a)\ D\ P\ b \\
\Box\ (ind\ H\ is\ \phi)\ D\ P\ (a, b) = \Sigma\ ((h : H) \to P\ (a\ h))\ \backslash\_ \to \Box\ \phi\ D\ P\ b \\
\\
map\Box : \{I : Set\}\ (\phi : Desc\ I)\ (D : I \to Set)\ (P : \{i : I\} \to D\ i \to Set) \\
\quad (p : \{i : I\}\ (d : D\ i) \to P\ d) \\
\quad \{i : I\}\ (v : [\![\ \phi\ ]\!]\ D\ i) \to \Box\ \phi\ D\ P\ v \\
map\Box\ (done\ i)\ \ D\ P\ p\ refl = \_ \\
map\Box\ (arg\ A\ \phi)\ \ D\ P\ p\ (a, b) = map\Box\ (\phi\ a)\ D\ P\ p\ b \\
map\Box\ (ind\ H\ is\ \phi)\ D\ P\ p\ (a, b) = (\backslash h \to p\ (a\ h)), map\Box\ \phi\ D\ P\ p\ b \\
\\
elim : \{I : Set\}\ (\phi : Desc\ I)\ (P : \{i : I\} \to \mu\ \phi\ i \to Set) \\
\quad (p : \{i : I\}\ (x : [\![\ \phi\ ]\!]\ (\mu\ \phi)\ i) \to \Box\ \phi\ (\mu\ \phi)\ P\ x \to P\ (intro\ x)) \\
\quad \{i : I\}\ (v : \mu\ \phi\ i) \to P\ v \\
elim\ \phi\ P\ p\ (intro\ v) = p\ v\ (map\Box\ \phi\ (\mu\ \phi)\ P\ (elim\ \phi\ P\ p)\ v)
\end{array}
$$

What does this buy us?

- Given a Type Theory with finite types and sigma types we can add datatypes by adding the rules for the universe described above.
- This new type theory is closed under the definition of new data-types, in some sense they are already present in the theory.
- In fact, we can go further, since the data types Desc and μ are themselves inductive families, we should be able to define them as codes in the Desc universe.
- But that's a bit circular, so we need a hierarchy of data universes $\mathrm{Desc}_i : \mathrm{Desc}_{i+1}$.
- In this way we only have to add rules to our TT for $[\![\_]\!]$ and elim.

Roadmap

Type Proliferation

- The properties and invariants we might want to specify are endless:
  - From Lists..
  - .. to Vectors ..
  - .. to Bounded Length Lists
  - .. to Sorted Lists ..
  - .. to Sorted Vectors ..
  - .. to Sorted, Bounded Lists
  - .. to Fresh Lists
  - ....
  - .. Profit?
- And each incarnation may need to be equipped with some notion of
  - Map
  - Concatenation
  - Fold
  - Filter

Generics

- Functional languages like Haskell already suffer from this problem (lite).
- There is a large research community pursuing a solution called *generic* or *polytypic* programming.
- A generic program is one that works on any of class of types, specialising its operation on the structure of type.
- Generic programming systems tend to be written as preprocessors, or make heavy use of experimental language systems.
- It turns out, what they really need is *universes*..

Universes for Generics

- Given a universe of data, a generic function is one that has this shape:

The shape of a generic function

$$\mathsf{foo} : \{\mathsf{u} : \mathsf{U}\} \to (\mathsf{x} : \mathsf{El}\ \mathsf{u}) \to \mathsf{T}\ \mathsf{u}\ \mathsf{x}$$

- Such a function will work for *any* type in the universe U, specialising its operation on the structure of the code u.
- In fact the function elim for the Desc universe we saw above, is a generic function.

Carving out useful universes

- We don't win just yet though, since the Desc universe is relatively large it supports very few generic programs.
- ..in fact only elim
- We don't need just one universe for generics, but rather many small universes, each supporting a different class of generic functions.
- Typically the functions we want to write determine the class of types the universe should capture.
- Looking at it in this way, we can see that Desc supports elim because it captures exactly those families which have a sound induction principle.