From Impredicativity to Induction in Dependent Type Theory

Aaron Stump
Computer Science
The University of Iowa
Iowa City, Iowa

Goal of the talk:

Explain how

- impredicative definitions of datatypes in type theory

$$3 \quad : \quad \underbrace{\forall X : \star.(X \to X) \to X \to X}_{Nat}$$

- can be refined into induction principles

$$3 \quad : \quad \forall X : Nat \to \star.(\forall n : Nat.\ X\ n \to X\ (S\ n)) \to X\ Z \to X\ \underline{3}$$

- using a new typing construct called *constructor-constrained recursive types*,
- and apply to the debate on the impredicativity of induction.

Outline

I. Impredicative definitions in type theory

II. Constructor-constrained recursive types

III. Is induction inherently impredicative?

I. Impredicative definitions in type theory

Impredicative definitions

“Whatever involves an apparent variable must not be among the possible values of that variable.” [Russell, 1906]

A definition of x is impredicative if it includes a quantification over a collection (possibly) containing x.

E.g., in a second-order logic:

$$Nat \;=\; \lambda n.\, \underline{\forall\phi}.\, (\forall x.\phi(x) \to \phi(S\,x)) \to \phi(0) \to \phi(n)$$

Similarly: $\{S \mid S \notin S\}$

Russell developed (ramified) type theory to prevent impredicativity

Impredicativity and constructivism

According to constructivism:

- Each new definition is viewed as creative
- There is no pre-existing Platonic universe

Impredicativity thus non-constructive:

- Cannot quantify over a totality containing x to create x

But this is philosophical constructivism.

A formal alternative: canonicity

- Types have canonical inhabitants
- Non-canonical terms must compute to canonical ones
- Not the case for classical principles like $A \vee \neg A$
- But impredicativity is compatible with canonicity

Impredicativity in type theory

Impredicative type theory: System F [Girard 1972, Reynolds 1974]

$$\textit{types } T \; ::= \; X \mid T \to T' \mid \forall X.\, T$$

Quantification $\forall X.T$ over all types is a type.

Alternative: predicative polymorphism:

- Types are stratified into levels $\star_0, \star_1, \ldots$
- Quantifications over level k are themselves in level at least $k+1$
- Unpleasant level calculations, level quantification, ordinal levels!

Church-encoded natural numbers in System F

Church-encoding: numbers are their own *iterators*

$$
\begin{array}{rcl}
0 & = & \lambda f.\, \lambda a.\, a \\
1 & = & \lambda f.\, \lambda a.\, f\ a \\
2 & = & \lambda f.\, \lambda a.\, f\ (f\ a) \\
 & & \quad \cdots \\
n & = & \lambda f.\, \lambda a.\, \underbrace{f \cdots (f}_{n}\ a) \\
 & & \quad \cdots
\end{array}
$$

Type in System F:

$$Nat = \forall X.(X \to X) \to X \to X$$

[Fortune, Leivant, O'Donnell 1983] [Böhm, Berarducci 1985] [Girard 1989]

Computing with Church-encoded numbers

Arithmetic operations:

$$
\begin{array}{lcl}
S & = & \lambda x.\lambda f.\lambda a.f\ (x\ f\ a) \\
add & = & \lambda x.\lambda y.\lambda f.\lambda a.\ x\ f\ (y\ f\ a) \\
mult & = & \lambda x.\lambda y.x\ (add\ y)\ 0 \\
exp & = & \lambda x.\lambda y.y\ (mult\ x)\ 1 \\
tetra & = & \lambda x.\lambda y.y\ (exp\ x)\ 1 \\
 & & \quad\quad\quad \cdots \\
ack & = & \ldots
\end{array}
$$

Impredicativity becomes essential after tetration:

Theorem (Leivant 1990)

The set of representable functions of <u>*predicative*</u> *System F is exactly $\mathcal{E}_4$ (Grzegorczyk class), the super-elementary functions.*

Alternative: Parigot encoding [Parigot 1988]

Addresses the problem of inefficient predecessor [Parigot 1989]

Define data as recursors, not iterators

$$\ulcorner n \urcorner = \lambda s.\lambda z.s \ulcorner n-1 \urcorner \cdots (s \ulcorner 1 \urcorner (s \ulcorner 0 \urcorner z))$$

For example, $\ulcorner 3 \urcorner$ is

$$\lambda s.\lambda z.s \ulcorner 2 \urcorner (s \ulcorner 1 \urcorner (s \ulcorner 0 \urcorner z))$$

Predecessor takes constant time

Typable in System F + positive-recursive types

$$Nat = \mu Nat.\, \forall X.\, (Nat \to X \to X) \to X \to X$$

Exponential-space normal forms, but not with graph sharing

$O(n^2)$-space encoding with efficient predecessor [Stump, Fu 2016]

From types to logic

Type theories correspond to logics (Curry-Howard)

- Martin-Löf type theory for constructive mathematics (e.g.)
- System F is second-order intuit. prop. logic (cf. QBF!)
- but lacks predicates, quantification over individuals

Calculus of Constructions: System F + dependent types $\Pi x : T.\, T'$ [Coquand, Huet 1988]

But: induction is not derivable [Geuvers 2001]

The metastasis of CC

1. Add inductive types as primitive
   - [Coquand, Paulin 1988], [Pfenning, Paulin 1989]
   - Calculus of Inductive Constructions [Werner 1994]
2. Layer a predicative hierarchy on top of the impredicative kind
   - Extended Calculus of Constructions [Luo 1990]
3. Coinductive types

Complex metatheory, some strange restrictions

Datatypes must be <u>*strictly*</u> *positive*

No large eliminations with impredicative datatypes

Type preservation fails with coinductive types

Still, Coq is alive and thriving

A puzzle

Impredicativity is enormously powerful, yet inadequate

Shouldn't there be some way to use functional encodings as a basis for type theory?

II. Constructor-constrained recursive types

Type-correctness proofs [Leivant]

Reasoning about functional programs and complexity classes associated with type disciplines, Leivant 1983.

Start with impredicative definitions in second-order logic:

$$Nat \;=\; \lambda n.\, \forall P.\, (\forall x. P\; x \to P\;(S\; x)) \to P\; 0 \to P\; n$$

Also constructors, recursive definitions of functional programs

$$\begin{array}{lcl} add\; Z\; y & = & y \\ add\; (S\; x)\; y & = & S\; (add\; x\; y) \end{array}$$

Key insight:

Proofs of type correctness $\simeq$ operations on lambda-encoded data

Programming with proofs: a second order type theory, Parigot 1988

A proof of $Nat\ x \to Nat\ (S\ x)$

Let Nat_P abbreviate $(\forall x. P\ x \to P\ (S\ x)) \to P\ 0 \to P\ x$

Let Γ abbreviate the context $Nat\ x,\ (\forall x. P\ x \to P\ (S\ x)),\ P\ 0$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \forall x. P\ x \to P\ (S\ x)}}{\Gamma \vdash P\ x \to P\ (S\ x)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash Nat\ x}}{\Gamma \vdash Nat_P\ x} \quad \dfrac{}{\Gamma \vdash \forall x. P\ x \to P\ (S\ x)}}{\Gamma \vdash P\ 0 \to P\ x} \qquad \dfrac{}{\Gamma \vdash P\ 0}}{\Gamma \vdash P\ x}}{\Gamma \vdash P\ (S\ x)}}{Nat\ x,\ (\forall x. P\ x \to P\ (S\ x)) \vdash P\ 0 \to P\ (S\ x)}}{Nat\ x \vdash Nat_P\ (S\ x)}}{Nat\ x \vdash Nat\ (S\ x)}}{\vdash Nat\ x \to Nat\ (S\ x)}
$$

Viewing this proof as a lambda-term (Curry-Howard):

$$\lambda x. \lambda f. \lambda a. f\ (x\ f\ a)$$

Type-correctness proof for recursive *add* is $\lambda x. \lambda y. \lambda f. \lambda a. x\ f\ (y\ f\ a)$

First step: lambda-encode

Instead of primitive constructors S and Z and operations like

$$\begin{array}{lcl} add\, Z\ y & = & y \\ add\, (S\ x)\ y & = & S\ (add\ x\ y) \end{array}$$

take

$$\begin{array}{lcl} Z & = & \lambda f.\lambda a.a \\ S & = & \lambda n.\lambda f.\lambda a.f\ (n\ f\ a) \\ add & = & \lambda x.\lambda y.\lambda f.\lambda a.x\ f\ (y\ f\ a) \end{array}$$

Recursive equations are satisfied modulo β

Now type-correctness proof for *add* is *add*

Similarly, type-correctness proof for 2 is 2:

$$2\ \ proves\ \ Nat\, 2$$

Next step: from logic to type theory

Trying to get to something like

$$2 \;:\; \mathit{Nat}\,2$$

Instead of second-order logic, use a type theory

But how can the type of a term mention that term?

Dependent intersection types

Dependent intersection types $x : A \cap B$ [Kopylov 2003]

I will use notation $\iota x : T.\, T'$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash t : [t/x]T'}{\Gamma \vdash t : \iota x : T.\, T'}$$

$$\frac{\Gamma \vdash t : \iota x : T'.T}{\Gamma \vdash t : T'} \qquad \frac{\Gamma \vdash t : \iota x : T'.T}{\Gamma \vdash t : [t/x]T}$$

This will be our tool to unify type-correctness proof and operation

Getting closer

Starting with

$$Nat = \lambda n. \forall P. (\forall x : Nat. P\ x \to P\ (S\ x)) \to P\ Z \to P\ n$$

We can move from a predicate to a type

$$Nat = \iota n. \forall P : Nat \to \star. (\forall x : Nat. P\ x \to P\ (S\ x)) \to P\ Z \to P\ n$$

Problems:

- The equation is circular
- What is the type of n? (untyped in [Fu and Stump 2014])
- How would we type S and Z?

A sequence of approximations

Let $\mathcal{U}$ be a universal type

$$\frac{FV(\lambda x.t) \subseteq dom(\Gamma)}{\Gamma \vdash \lambda x.t : \mathcal{U}}$$

Now define at the meta-level:

$$\begin{array}{lcl} Nat_0 & := & \mathcal{U} \\ Nat_{k+1} & := & \iota n : Nat_k . \forall P : Nat_k \to \star . \\ & & (\forall n : Nat_k . P\ n \to P\ (S\ n)) \to P\ Z \to P\ n \end{array}$$

Nat_{k+1} lets us do induction with level k predicates P

Also, for all k:

$$\begin{array}{lcl} Z & : & Nat_k \\ S & : & Nat_k \to Nat_k \end{array}$$

Final step: take the limit

$$\nu Nat : \star \mid S : Nat \to Nat,\ Z : Nat .$$
$$\iota n : Nat.\ \forall P : Nat \to \star.\ (\forall n : Nat.\ P\ n \to P\ (S\ n)) \to P\ Z \to P\ n$$

Constructor-constrained recursive type

Take greatest lower bound of the descending sequence

ν-bound variable must be used only positively (monotonicity)

Constructor typings must hold initially and be preserved

Positivity also required for constructors' argument types

Lattice-theoretic semantics

$$\begin{array}{rcl}\mathcal{L} & = & \{\lambda x.t \mid FV(\lambda x.t) = \varnothing\} \\ \mathcal{R} & = & \{[S]_{c\beta} \mid S \subseteq \mathcal{L}\}\end{array}$$

$$\begin{array}{rcl}[\![\nu X : \kappa \mid \Theta.T]\!]_{\sigma,\rho} & = & q, \textit{ where} \\ & & q = \cap_{\kappa,\sigma,\rho}\{F^n(\top_{\kappa,\sigma,\rho}) \mid n \in \mathbb{N}\} \text{ and} \\ & & F = (S \in [\![\kappa]\!]_{\sigma,\rho} \mapsto [\![T]\!]_{\sigma,\rho[X \mapsto S]})\}; \\ & & \text{if } F(q) = q\end{array}$$

Theorem (Soundness)

If $(\sigma, \rho) \in [\![\Gamma]\!]$, *then*

1. *If* $\Gamma \vdash \kappa : \Box$, *then* $[\![\kappa]\!]_{\sigma,\rho}$ *is defined.*
2. *If* $\Gamma \vdash T : \kappa$, *then* $[\![T]\!]_{\sigma,\rho} \in [\![\kappa]\!]_{\sigma,\rho}$.
3. *If* $\Gamma \vdash t : T$, *then* $[\sigma t]_{c\beta} \in [\![T]\!]_{\sigma,\rho}$ *and* $[\![T]\!]_{\sigma,\rho} \in \mathcal{R}$.

Corollary (Consistency)

$\forall X : \star. X$ *is uninhabited.*

What have we done?

$$\begin{array}{l}\nu\, Nat : \star \mid S : Nat \to Nat,\ Z : Nat\ . \\ \quad \iota n : Nat.\ \forall P : Nat \to \star.\ (\forall n : Nat.\ P\ n \to P\ (S\ n)) \to P\ Z \to P\ n\end{array}$$

Each number proves its own induction principle.

$\underline{2\ :\ Nat}$ is equivalent to

$$2\ :\ \forall P : Nat \to \star.\ (\forall n : Nat.\ P\ n \to P\ (S\ n)) \to P\ Z \to P\ \boxed{2}$$

Dependent type theory based on functional encodings

Define operations on *Nat* (e.g., *add*)

Using an equality type, can start to develop number theory.

$$\Pi x : Nat.\, \Pi y : Nat.\ add\ x\ y\ \simeq\ add\ y\ x$$

Proved by induction on x

So we apply x to the step and base cases.

III. Is induction inherently impredicative?

Returning to philosophy

Philosophical constructivists (should) oppose impredicativity

- If definition of x is creative, then cannot appeal to set containing x
- So constructivists cannot accept

$$Nat = \lambda n. \forall P. (\forall x : Nat. P\ x \rightarrow P\ (S\ x)) \rightarrow P\ Z \rightarrow P\ n$$

- They do wish to accept induction, however

An alternative is to give semantics for *Nat* via an induction rule

$$\frac{Nat\ x \quad \forall x. P\ x \rightarrow P\ (S\ x) \quad P\ 0}{P\ x}$$

This avoids explicit impredicative quantification

Objection of Parsons

The Impredicativity of Induction, [Parsons 1983/1992]

$$\frac{\mathit{Nat}\ x \quad \forall x. P\ x \to P\ (S\ x) \quad P\ 0}{P\ x}$$

Suppose induction is taken as explaining *Nat*

Then this explanation is still impredicative:

Predicates P in the induction rule can include *Nat*

So why reject impredicative quantification but accept this?

"Some impredicativity is inevitable in mathematical concept formation."
[Parsons 1992]

Alternative is to deny that induction explains *Nat*

View of Thorsten Altenkirch [private communication]

We understand numbers intuitively, and induction is a consequence

Does this ν approach shed any light?

We have seen how to pass from type-theoretic impredicativity

$$\forall X : \star.(X \to X) \to X \to X$$

to a definition making numbers their own inductions

$$\begin{array}{l} \nu Nat : \star \mid S \ : Nat \to Nat,\ Z \ : \ Nat\ . \\ \quad \iota n : Nat.\ \forall P : Nat \to \star.\ (\forall n : Nat.\ P\ n \to P\ (S\ n)) \to P\ Z \to P\ n \end{array}$$

So we have:

$$n \ : \ \forall P : Nat \to \star.\ (\forall n : Nat.\ P\ n \to P\ (S\ n)) \to P\ Z \to P\ \boxed{\mathrm{n}}$$

By using lambda-encoding, no need for primitive 0 and S

But hard to see how to turn the impredicative type into a rule

Contrasting views

Constructivist:

Reject impredicativity

Take numbers as given

Induction is a consequence

Induction is a rule

Impredicativist:

Embrace impredicativity

Define numbers

Induction is essential

Induction is a type

Do I have to be a Platonist to use your theory?

Do I have to be a Platonist to use your theory?

No.

Do I have to be a Platonist to use your theory?

No.

Impredicativity does presuppose an existing Platonic universe

But we are only theorizing about such a universe

Nothing says our theory need describe our own universe

But then!

How can the theory be useful?

But then!

How can the theory be useful?

My view: our universe finitely approximates that of the theory

A formal analogy

Every term typable in System F is normalizing

- "Sound for normalization"
- Not complete, though
- Proof is complex (proof-theoretically)
- $[\![\forall X.T]\!]_\rho = \bigcap \{[\![T]\!]_{\rho[X\mapsto R]} \mid R \in \mathcal{R}\}$

Contrast with type systems based on finite intersection types

- Sound and complete for normalization!
- Proof is easy (see Barendregt "Lambda Calculus with Types", 2010)

Why the difference?

- Finite intersections: types needed for the (finite) reduction graph
- Infinite intersections: types needed for all possible calling contexts

Conclusion

We have seen how

- impredicative definitions of datatypes in type theory

$$3 \quad : \quad \underbrace{\forall X : \star . (X \to X) \to X \to X}_{Nat}$$

- can be refined into induction principles

$$3 \quad : \quad \forall X : Nat \to \star . (\forall n : Nat.\ X\ n \to X\ (S\ n)) \to X\ Z \to X\ \underline{3}$$

- using a new typing construct called *constructor-constrained recursive types*,
- and considered in light of debate on impredicativity of induction.

The Calculus of Dependent Lambda Eliminations

A full type theory based on these ideas

Includes also an operator to lift *simply typed* terms to the type level

$$\uparrow_{(\star\to\star)\to\star\to\star}(\lambda s.\lambda z.s\ z)\ \simeq\ \lambda S:\star\to\star.\lambda Z:\star.S\ Z$$

Supports computing a predicate by natural-number recursion (e.g.)

Denotational semantics for types, consistency proof

See my web page for manuscript under review

Tomorrow will talk about an implementation Cedille, applications

The Calculus of Dependent Lambda Eliminations

A full type theory based on these ideas

Includes also an operator to lift *simply typed* terms to the type level

$$\uparrow_{(\star\to\star)\to\star\to\star}(\lambda s.\lambda z.s\ z)\ \simeq\ \lambda S:\star\to\star.\lambda Z:\star.S\ Z$$

Supports computing a predicate by natural-number recursion (e.g.)

Denotational semantics for types, consistency proof

See my web page for manuscript under review

Tomorrow will talk about an implementation Cedille, applications

Thanks!