INTRODUCTION TO

DEPENDENT TYPES

Carl Factora
cfactora@indiana.edu

WE WANT TO ANSWER THE QUESTION:

WHAT IS A DEPENDENT TYPE?

WIR MÜSSEN WISSEN. WIR WERDEN WISSEN. [D. HILBERT]

We must know. We will know.

- Brief Intro to Intuitionistic Logic
- Curry-Howard-DeBruijn Isomorphism
- Some Proofs
- the λ-Cube ($\lambda\!\rightarrow$ to λC)
- Extra: “All You Need Is…”

EITHER MATHEMATICS IS TOO BIG FOR THE HUMAN MIND, OR THE HUMAN MIND IS MORE THAN A MACHINE. [K. GÖDEL]

- "A logic where people matter…" [R. Harper]
- "… where double-negation, excluded third, and indirect proof are the casualties." [C. Factora]
- Works on the idea that **not every logical expression** is either **true** or **false**
- Negation and Disjunction are quite a bit different from their Classical Logic counterparts

EITHER MATHEMATICS IS TOO BIG FOR THE HUMAN MIND, OR THE HUMAN MIND IS MORE THAN A MACHINE. [K. GÖDEL]

- *Negation* (e.g. ¬A) reads "*We cannot prove* A."
  - ¬¬A reads "*We cannot prove that we cannot prove* A."
  - *Not* equivalent to A, which is read "*We have a proof of* A."
  - We no longer have access to *indirect proof* methods.
- *Disjunction* (e.g. A v B) must be treated with caution.
  - A v ¬A is not a logical tautology

$$\frac{A \qquad \neg A}{\bot}\,(\bot\text{-intro})^{**} \qquad \frac{A}{A \vee B}\,(\vee\text{-intro-l}) \qquad \frac{B}{A \vee B}\,(\vee\text{-intro-r})$$

$$\frac{\bot}{A}\,(\bot\text{-elim}) \qquad \frac{A \vee B \quad A \to C \quad B \to C}{C}\,(\vee\text{-elim})$$

**mind the lack of orange on this slide*

***special case of* modus ponens

TIME FOR A LITTLE BIT OF ORANGE

- Q : What is a proposition?

- **Q** : What is a proposition?

- **A** : Something we can *prove*.

- **Q** : What is a proposition?

- **A** : Something we can *prove*.

$$((A \to B) \land A) \to B$$

- **Q** : What is a type?
- **A** : Something we can *inhabit*.

- **Q** : What is a type?
- **A** : Something we can *inhabit*.

$((A \to B) \land A) \to B$

- Q : What is a proof?

- **Q** : What is a proof?
- **A** : Something that *proves* a *proposition*.

- **Q** : What is a proof?

- **A** : Something that *proves* a *proposition*.

$$((A \to B) \wedge A) \to B$$

CP, ^-elimination, *modus ponens*

- **Q** : What is a term?

- **A** : Something that *inhabits* a type.

- **Q** : What is a **term**?

- **A** : Something that *inhabits* a **type**.

Assume, Pair Deconstruction, Apply

$\lambda c{:}(A \to B) \wedge A \,.\, (\mathrm{fst}(c))(\mathrm{snd}(c)) : ((A \to B) \wedge A) \to B$

What does the ":" mean?

$\lambda c{:}(A \rightarrow B) \wedge B \,.\, (\mathrm{fst}(c))(\mathrm{snd}(c)) : ((A \rightarrow B) \wedge A) \rightarrow B$

What does the ":" mean?

$\lambda c{:}(A \rightarrow B) \wedge B\,.\,(fst(c))(snd(c)) : ((A \rightarrow B) \wedge A) \rightarrow B$

":" is read "is a proof of"

What does the ":" mean?

$\lambda c{:}(A \rightarrow B) \wedge B\,.\,(\mathrm{fst}(c))(\mathrm{snd}(c)) : ((A \rightarrow B) \wedge A) \rightarrow B$

":" is read "is of type"

What does the ":" mean?

$\lambda c{:}(A \to B) \wedge B \,.\, (fst(c))(snd(c)) : ((A \to B) \wedge A) \to B$

":" is read "is a proof of"

":" is read "is of type"

NOW, WHY DIDN'T I THINK OF THAT? [P. MARTIN-LÖF]

- *PAT Interpretation*: propositions as types, proofs as terms

Proposition = Type

Proof = Term

NOW, WHY DIDN'T I THINK OF THAT? [P. MARTIN-LÖF]

- *PAT Interpretation*: propositions as types, proofs as terms

Proposition = Type

Proof = Term

Simplification = Evaluation

TIME FOR SOME PROOFS

$A \land B \to B \land A$

$$\vdots$$

$$?_1 \quad : \quad A \land B \to B \land A$$

$A \wedge B \to B \wedge A$

$$[c : A \wedge B]$$

$$?_2 \quad : \quad B \wedge A$$

$$\frac{?_2 : B \wedge A}{?_1 : A \wedge B \to B \wedge A}\,(c)$$

A∧B→B∧A

$$\dfrac{\begin{array}{c}[c : A \wedge B] \\ \\ (\mathrm{snd}(c), \mathrm{fst}(c)) \quad : \quad B \wedge A\end{array}}{?_1 \quad : \quad A \wedge B \rightarrow B \wedge A}\;(c)$$

A∧B→B∧A

$$\frac{\begin{array}{c}[c : A \wedge B] \\ \vdots \\ (\mathrm{snd}(c), \mathrm{fst}(c)) \ : \ B \wedge A\end{array}}{\lambda c.\, (\mathrm{snd}(c), \mathrm{fst}(c)) \ : \ A \wedge B \rightarrow B \wedge A}\,(c)$$

AND NOW, WITH QUANTIFIERS

$\neg\exists x : S\ .P_x \to \forall x : S\ .\neg P_x$

$$\cfrac{\cfrac{\cfrac{\vdots}{\vdots}}{\vdots}}{?_1 \quad : \quad \neg\exists x : S\ .P_x \to \forall x : S\ .\neg P_x}$$

We use the notation Px to say that P is a proposition over *x*.

$\neg\exists x : S\,.P_x \rightarrow \forall x : S\,.\neg P_x$

$$[ne : \neg\exists x : S\,.P_x]$$

$$\vdots$$

$$\vdots$$

$$\dfrac{?_2 \;:\; \forall x : S\,.\neg P_x}{?_1 \;:\; \neg\exists x : S\,.P_x \rightarrow \forall x : S\,.\neg P_x}\,(ne)$$

$\neg\exists x : S . P_x \rightarrow \forall x : S . \neg P_x$

$$[ne : \neg\exists x : S . P_x] \qquad [x : S]$$

$$\vdots$$

$$\frac{\dfrac{\dfrac{?_3 : \neg P_x}{?_2 : \forall x : S . \neg P_x}\,(x)}{?_1 : \neg\exists x : S . P_x \rightarrow \forall x : S . \neg P_x}\,(ne)}{}$$

$\neg\exists x : S\,.P_x \to \forall x : S\,.\neg P_x$

$$[ne:\neg\exists x : S\,.P_x] \quad [x : S] \quad [p : P_x]$$

$$\dfrac{\dfrac{\dfrac{?_3 : \bot}{?_3 : \neg P_x}(p)}{?_2 : \forall x : S\,.\neg P_x}(x)}{?_1 : \neg\exists x : S\,.P_x \to \forall x : S\,.\neg P_x}(ne)$$

$\neg\exists x : S . P_x \to \forall x : S . \neg P_x$

$$[ne : \neg\exists x : S . P_x] \qquad [x : S] \qquad [p : P_x]$$

$$\begin{array}{ccc|l}
ne(x,p) & : & \bot & \\
\hline
?_3 & : & \neg P_x & (p) \\
\hline
?_2 & : & \forall x : S . \neg P_x & (x) \\
\hline
?_1 & : & \neg\exists x : S . P_x \to \forall x : S . \neg P_x & (ne)
\end{array}$$

$\neg\exists x : S\ .P_x \rightarrow \forall x : S\ .\neg P_x$

$$[ne : \neg\exists x : S\ .P_x] \qquad [x : S] \qquad [p : P_x]$$

$$\begin{array}{ccc}
ne(x,p) & : & \bot \\
\hline
\lambda p.ne(x,p) & : & \neg P_x \\
\hline
?_2 & : & \forall x : S\ .\neg P_x \\
\hline
?_1 & : & \neg\exists x : S\ .P_x \rightarrow \forall x : S\ .\neg P_x
\end{array}$$

The three rule lines are labelled, from top to bottom: (p), (x), (ne)

$\neg\exists x : S . P_x \rightarrow \forall x : S . \neg P_x$

$$[ne : \neg\exists x : S . P_x] \qquad [x : S] \qquad [p : P_x]$$

$$\dfrac{\dfrac{\dfrac{ne(x,p) \;:\; \bot}{\lambda p.ne(x,p) \;:\; \neg P_x}\,(p)}{\lambda x.\lambda p.ne(x,p) \;:\; \forall x : S . \neg P_x}\,(x)}{?_1 \;:\; \neg\exists x : S . P_x \rightarrow \forall x : S . \neg P_x}\,(ne)$$

$\neg\exists x : S\,.P_x \rightarrow \forall x : S\,.\neg P_x$

$$[ne{:}\neg\exists x : S\,.P_x] \quad [x : S] \quad [p : P_x]$$

$$\dfrac{\dfrac{\dfrac{ne(x,p) \;:\; \bot}{\lambda p.ne(x,p) \;:\; \neg P_x}(p)}{\lambda x.\lambda p.ne(x,p) \;:\; \forall x : S\,.\neg P_x}(x)}{\lambda ne.\lambda x.\lambda p.ne(x,p) \;:\; \neg\exists x : S\,.P_x \rightarrow \forall x : S\,.\neg P_x}(ne)$$

PI AND SIGMA

Π represents a *generalized* function. This is essentially what allows us to model the behavior the universal quantifier, ∀.

Σ represents a *generalized* pair. This allows us to model the behavior of the existential quantifier, ∃.

PI AND SIGMA

$\Pi x{:}A\,.\,B$ is equivalent to $A \to B$, *iff* x does not appear free in B.

$\Sigma x{:}A\,.\,B$ is equivalent to $A \wedge B$, *iff* x does not appear free in B.

PI AND SIGMA

$\neg\Sigma x : S\,.P_x \to \Pi x : S\,.\neg P_x$

$$[ne : \neg\Sigma x : S\,.P_x] \quad [x : S] \quad [p : P_x]$$

$$
\begin{array}{ccc}
ne(x,p) & : & \perp \\
\hline
\lambda p.ne(x,p) & : & \neg P_x \\
\hline
\lambda x.\lambda p.ne(x,p) & : & \Pi x : S\,.\neg P_x \\
\hline
\lambda ne.\lambda x.\lambda p.ne(x,p) & : & \neg\Sigma x : S\,.P_x \to \Pi x : S\,.\neg P_x
\end{array}
$$

(p)

(x)

(ne)

Idris

"Hello World!" : String

DEPENDENT TYPES

The concept of *dependent types* refers to the existence of a *dependency* between types and terms, generally excluding *terms depending on terms*.

In this case, the particular dependency refers to "meaning" (i.e., the meaning of types).

DEPENDENT TYPES

IT'S WRONG TO CALL THE Λ-CALCULUS THE UNIVERSAL PROGRAMMING LANGUAGE… THAT'S TOO LIMITING.

[P. WADLER]

$x \;||\; \lambda x.\Lambda \;||\; (\Lambda\ \Lambda)$

IT'S WRONG TO CALL THE Λ-CALCULUS THE UNIVERSAL PROGRAMMING LANGUAGE… THAT'S TOO LIMITING.

[P. WADLER]

Abstractions create dependency.

- Possible system variations:
  - *terms* depending on *terms* (danger; no types)
  - *terms* depending on *types*
  - *types* depending on *types*
  - *types* depending on *terms*

(Ω)

$$((\lambda x.xx)\ (\lambda x.xx))$$

[⊥] 1937 – forever

THE HALTING PROBLEM “SOLVED”*

*not really

Ω̋

$$((\lambda\, x{:}A \to B\,.\; x\; x)$$
$$(\lambda\, x{:}A \to B\,.\; x\; x))$$

[Type-check Error] R.I.P.

- Possible (type) system variations:
  - *terms* depending on *terms* (a basic type system)
  - *terms* depending on *types*
  - *types* depending on *types*
  - *types* depending on *terms*
- How about a system with all of the above? How can we model the relationship they have with each other?

THE LAMBDA CUBE

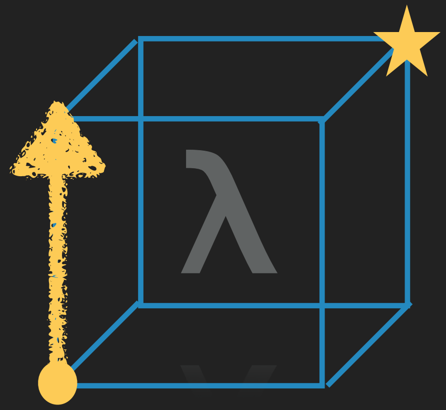
Another application…

$\lambda x{:}\alpha\,.\,x$

TERMS DEPENDING ON TERMS

THERE MIGHT BE OTHER APPLICATIONS… [A. CHURCH]

- No more infinite loops
- *Lambda Term* (Λ) = x:T || λx:T.Λ || (Λ Λ)
- *Simple Type* (T) = α || T → T
- Examples:
  - λx:α.λy:β.x : α→β→α [K-Combinator]
  - (λx:α.λy:β.x) a b : α



$$\lambda\alpha{:}*.\ \lambda x{:}\alpha.\ x$$

TERMS DEPENDING ON TYPES

THERE MIGHT BE OTHER APPLICATIONS… [A. CHURCH]

- Addition of *terms depending on types*
- *Lambda Term* (Λ) = $x{:}T \mid\mid \lambda x{:}T.\Lambda \mid\mid (\Lambda\ \Lambda) \mid\mid \lambda\alpha{:}*.\Lambda \mid\mid (\Lambda\ T)$
- *Type* (T) = $\alpha \mid\mid T \rightarrow T \mid\mid \Pi\alpha{:}*.\,T$
- Examples:
  - $\lambda\alpha{:}*.\lambda\beta{:}*.\lambda x{:}\alpha.\lambda y{:}\beta.x : \Pi\alpha{:}*.\ \Pi\beta{:}*.\ \alpha\rightarrow\beta\rightarrow\alpha$ [Polymorphic K-Combinator]
  - $(\lambda\alpha{:}*.\lambda\beta{:}*.\lambda x{:}\alpha.\lambda y{:}\beta.x)\ t_1\ t_2\ a\ b : t_1$

$$\lambda\alpha{:}* \,.\, \alpha \to \alpha$$

TYPES DEPENDING ON TYPES

THERE MIGHT BE OTHER APPLICATIONS… [A. CHURCH]

- Addition of *types depending on types*
- *Lambda Term* (Λ) = $x{:}T \;||\; \lambda x{:}T.\Lambda \;||\; (\Lambda\ \Lambda)$
- *Type* (T) = $\alpha \;||\; T \rightarrow T \;||\; \lambda\alpha{:}*.\ T \;||\; (T\ T)$
- Example:
  - $\lambda\alpha{:}*.\ \alpha\rightarrow\alpha$ [cannot be inhabited]
  - $(\lambda\alpha{:}*.\ \alpha\rightarrow\alpha)\ t_1 =_\beta (t_1\rightarrow t_1)$ [can be inhabited]

$$\Pi x{:}\alpha \,.\, P_x$$

TYPES DEPENDING ON TERMS

THERE MIGHT BE OTHER APPLICATIONS… [A. CHURCH]

- Addition of *types depending on terms*
- *Lambda Term* (Λ) = $x{:}T \,\|\, \lambda x{:}T.\Lambda \,\|\, (\Lambda\ \Lambda)$
- *Type* (T) = $\alpha \,\|\, T \rightarrow T \,\|\, \Pi x{:}T.\,T$
- Q: "So, what do *types depending on terms* give us?"
- A: "$\Pi x{:}\alpha.\ P_x$, (i.e., we get quantified propositions)."

We use the notation Px to say that x can appear free in P

∀ OF THE ABOVE

THERE MIGHT BE OTHER APPLICATIONS… [A. CHURCH]

- How to get to λC (start with λ→):
  - Move "up" :: λ2
  - Move "right" :: λ<u>ω</u>
  - Move "in" :: λP
- In this talk, we took one path:
  - λ→ *to* λ2 *to* λω (i.e., λ<u>ω</u>+ λ2) *to* λC (i.e., λω+ λP)

CONCLUSION

terms depending on *terms*	→	simply-typed λ-calculus
terms depending on *types*	→	polymorphic λ-calculus (System F)
types depending on *types*	→	higher-order λ-calculus
types depending on *terms*	→	predicate λ-calculus

CONCLUSION

terms depending on *terms*	→	simply-typed λ-calculus
terms depending on *types*	→	polymorphic terms
types depending on *types*	→	type constructors (polymorphic types)
types depending on *terms*	→	generalized types

CONCLUSION

[1] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics* 1984: Elsevier Science.

[2] Henk Barendregt. *Lambda Calculi with types* 1992: Handbook of Logic in Computer Science, pp. 117-309, Oxford University Press.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: CoqÁrt: the Calculus of Inductive Constructions* 2004: Springer.

[4] Alonzo Church. *A formulation of the simple theory of types* 1940: Journal of Symbolic Logic, 5, pp. 56-68.

[5] Thierry Coquand and Gérard Huet. *The Calculus of Constructions* 1988: Information and Computation, 76, pp. 95-120.

[6] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur* 1972: PhD thesis, Université Paris VII.

[7] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types* 1989: Cambridge University Press, pp. 82-86.

[8] William Howard. *The formulae-as-types notion of construction* 1980: Seldin and Hindley, pp. 479-490.

[9] Per Martin-Löf. *Intuitionistic Type Theory* 1980: Bibliopolis

[10] Rob Nederpelt and Herman Geuvers. *Type Theory and Formal Proof* 2014: Cambridge University Press.

[11] Luis Sanchis. *Functionals defined by Recursion* 1967: Notre Dame Journal of Formal Logic VIII, pp. 161-174.

FURTHER READING

SO, WHAT YOU'RE SAYING IS, ALL I NEED IS PI?

[D.P. Friedman]

Π IS LIKE THE TWISTED SISTER OF Λ. [C. FACTORA]

- $\bot$ :: $\Pi\alpha{:}* \,.\, \alpha$
- $\neg$ :: $\lambda\alpha{:}* \,.\, \alpha\to\bot$
- $\to$ :: $\lambda\alpha{:}* \,.\, \lambda\beta{:}* \,.\, \alpha\to\beta$
- $\wedge$ :: $\lambda\alpha{:}* \,.\, \lambda\beta{:}* \,.\, \Pi\gamma{:}* \,.\, (\alpha\to\beta\to\gamma)\to\gamma$
- $\vee$ :: $\lambda\alpha{:}* \,.\, \lambda\beta{:}* \,.\, \Pi\gamma{:}* \,.\, (\alpha\to\gamma)\to(\beta\to\gamma)\to\gamma$
- $\forall$ :: $\lambda\sigma{:}* \,.\, \lambda\rho{:}\sigma\to* \,.\, \Pi x{:}\sigma \,.\, \rho_x$
- $\exists$ :: $\lambda\sigma{:}* \,.\, \lambda\rho{:}\sigma\to* \,.\, \Pi\alpha{:}* \,.\, (\Pi x{:}\sigma \,.\, \rho_x\to\alpha)\to\alpha$


All you need is Π
All you need is Π
All you need is Π
Π is all you need

WOULD YOU LIKE SOME PI?

$A \wedge B \to B \wedge A$

$$\frac{\begin{array}{c}[c : A \wedge B] \\ \\ (\mathrm{snd}(c), \mathrm{fst}(c)) \quad : \quad B \wedge A\end{array}}{\lambda c.\,(\mathrm{snd}(c), \mathrm{fst}(c)) \quad : \quad A \wedge B \to B \wedge A}\,(c)$$

Proof. $\Pi C : * \,.\, (A \to B \to C) \to C \to \Pi C : * \,.\, (B \to A \to C) \to C$

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}[e : \Pi C : * \,.\, (A \to B \to C) \to C], [c : *], [f : B \to A \to C] \\ e\,c\,(\lambda(a, b)\,.\,f\,b\,a) : C\end{array}}{\lambda f\,.\,e\,c(\lambda(a, b)\,.\,f\,b\,a) : (B \to A \to C) \to C}\,(f)}{\lambda(c, f)\,.\,e\,c(\lambda(a, b)\,.\,f\,b\,a) : \Pi C : * \,.\, (B \to A \to C) \to C}\,(c)}{\lambda(e, c, f)\,.\,e\,c(\lambda(a, b)\,.\,f\,b\,a) : \Pi C : * \,.\, (A \to B \to C) \to C \to \Pi C : * \,.\, (B \to A \to C) \to C}\,(e)$$

$\neg\Sigma x : S\,.P_x \rightarrow \Pi x : S\,.\neg P_x$

$$[ne : \neg\Sigma x : S\,.P_x] \quad [x : S] \quad [p : P_x]$$

$$\dfrac{\dfrac{\dfrac{ne(x,p) : \bot}{\lambda p.ne(x,p) : \neg P_x}\,(p)}{\lambda x.\lambda p.ne(x,p) : \Pi x : S\,.\neg P_x}\,(x)}{\lambda ne.\lambda x.\lambda p.ne(x,p) : \neg\Sigma x : S\,.P_x \rightarrow \Pi x : S\,.\neg P_x}\,(ne)$$

Proof. $\neg(\Pi a : * . (\Pi x : S . P_x \to a)) \to \Pi x : S . \neg P_x$

$$
\dfrac{\dfrac{\dfrac{\begin{array}{c}[ne : \neg(\Pi a : * . (\Pi x : S . P_x \to a) \to a)], [x : S], [p : P_x] \\ ne(\lambda a . \lambda y : \Pi x : S . P_x \to \alpha . yxp) : \bot\end{array}}{\lambda p . ne(\lambda a . \lambda y : \Pi x : S . P_x \to \alpha . yxp) : \neg P_x}\,(p)}{\lambda x . \lambda p . ne(\lambda a . \lambda y : \Pi x : S . P_x \to \alpha . yxp) : \Pi x : S . \neg P_x}\,(x)}{\lambda ne . \lambda x . \lambda p . ne(\lambda a . \lambda y : \Pi x : S . P_x \to \alpha . yxp) : \neg \Sigma x : S . P_x \to \Pi x : S . \neg P_x}\,(ne)
$$

THANKS FOR LISTENING! QUESTIONS?

Carl Factora | cfactora@indiana.edu