Propositions and Types, Proofs and Programs

Part V: Extensions

Ranald Clouston

School of Computing

Australian National University

ANU Logic Summer School 2023

Gödel's System T

We mentioned previously that we would like our base types to include useful types such as Bool and Nat.

- Including introduction and elimination rules
- System T extends STLC with exactly these types

These types are not interesting from a traditional logical point of view

- Logically equivalent to each other, and to any other theorem, e.g. $\bot \to \bot$
- But logical equivalence does not imply isomorphism
- This is the **proof relevant** view of logic which cares about which proofs inhabit each type, not mere inhabitation.

Booleans

Introduction and elimination:

$$\frac{}{\Gamma \vdash \texttt{True}:\texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{False}:\texttt{Bool}}$$

$$\frac{\Gamma \vdash b:\texttt{Bool} \quad \Gamma \vdash t:C \quad \Gamma \vdash u:C}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ u:C}$$

Reduction rules:

- $\texttt{if True then t else u} \mapsto \texttt{t}$ and $\texttt{if False then t else u} \mapsto \texttt{u}$

Booleans Discussed

We can therefore write programs with Booleans like

- $\lambda x^{\texttt{Bool}}.\texttt{if } x \texttt{ then False else True}$
- $\lambda x^{\texttt{Bool}}.\lambda y^{\texttt{Bool}}.\texttt{if } x \texttt{ then } y \texttt{ else False}$

We could instead encode Bool in STLC

- $(0 \to 0) + (0 \to 0)$
- True as $\iota^1(\lambda x^0.x)$ and False as $\iota^2(\lambda x^0.x)$
- if…then…else left as an exercise

But Nat, which is not finite, will require us to go beyond STLC

Natural Numbers

Introduction and elimination:

$$\frac{}{\Gamma \vdash \mathtt{zero}:\mathtt{Nat}} \qquad \frac{\Gamma \vdash n:\mathtt{Nat}}{\Gamma \vdash \mathtt{suc}\ n:\mathtt{Nat}}$$

$$\frac{\Gamma \vdash n:\mathtt{Nat} \quad \Gamma \vdash t:A \quad \Gamma \vdash f:A\to(\mathtt{Nat}\to A)}{\Gamma \vdash \mathtt{rec}(n,t,f):A}$$

`rec` corresponds to **primitive recursion**

- `n` is a counter; `t` is the base (zero) case; `f` is the recursive step

Natural Numbers - Examples

Test for `zero`:

$\lambda x^{Nat}.rec(x,\ True,\ \lambda z^{Bool}.\lambda z'^{Nat}.False)$

Plus:

$\lambda x^{Nat}.\lambda y^{Nat}.rec(x,\ y,\ \lambda z^{Nat}.\lambda z'^{Nat}.succ\ z)$

Predecessor (with `zero` mapped to `zero`):

$\lambda x^{Nat}.rec(x,\ zero,\ \lambda z^{Nat}.\lambda z'^{Nat}.z')$

Factorial (definition of `multiply : Nat→(Nat→Nat)` left as an exercise):

$\lambda x^{Nat}.rec(x,\ suc\ zero,\ \lambda z^{Nat}.\lambda z'^{Nat}.multiply\ z\ (suc\ z'))$

Natural Numbers - Reduction

Beta rules (we will ignore other rules e.g. for the subformula property):

- `rec(zero,t,f) ↦ t`
- `rec(suc n,t,f) ↦ (f (rec(n,t,f))) n`

Some examples from the last slide:

- `rec(zero,True,λz.λz'.False) ↦ True`
- `rec(suc n,True,λz.λz'.False)`
  `↦ ((λz.λz'.False)(rec(n,True,λz.λz'.False)))n ↦* False`
- `λx.rec(suc n,zero,λz.λz'.z’)`
  `↦ ((λz.λz'.z')(rec(n,zero,λz.λz'.z')))n ↦* n`

System T and Termination

We must give up at least one of the three:

- Strong normalisation
- Decidable type-checking
- Turing completeness (all terminating algorithms representable)

System T with `Nat` types is much more expressive than STLC but is likewise *not* Turing-complete.

Ackermann's function:

`A(0,n) = n+1; A(m+1,0) = A(m,1); A(m+1,n+1) = A(m,A(m+1,n))`

- Expressible in the untyped lambda calculus but not in System T.

Extending System T to Turing Completeness

To make System T a universal model of computation we need to strengthen the notion of recursion, and give up strong normalisation.

This is a perfectly sensible thing to do for a programming language but destroys the formal relationship with logic.

- *All* types are inhabited by programs that fail to terminate;
- So the corresponding logic is inconsistent.
- However as long as we attempt to write terminating programs only, we can take inspiration from Curry-Howard even in a Turing complete language;
- Indeed this has been a major source of design ideas for real programming languages.

Dependency

In the STLC and System T there is one sort of variable:

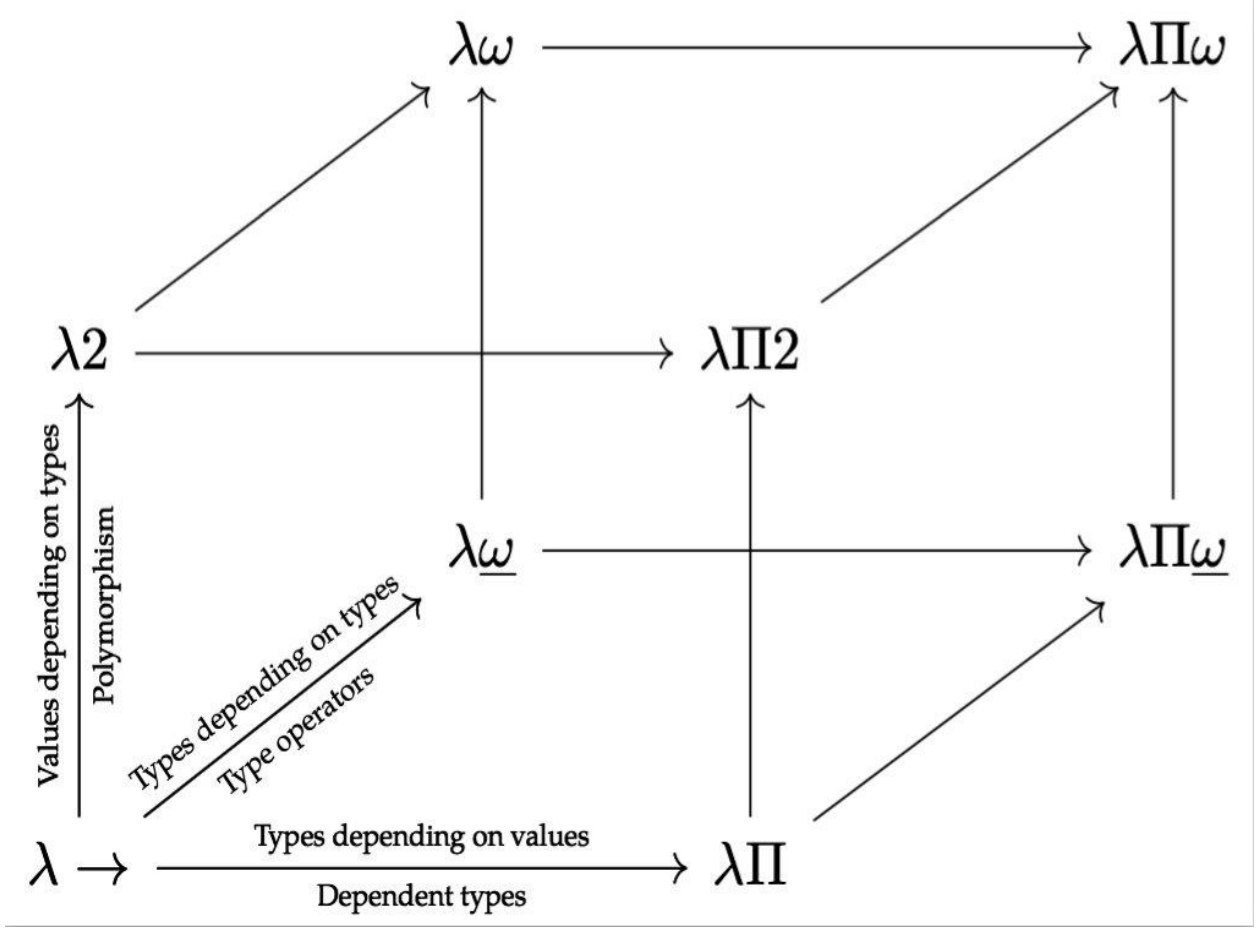
- Variables **appear in** terms; and can be **replaced by** terms
- We could say that **terms** depend on (unknown) **terms**.

But there are three other options that could be considered:

- Terms depend on types;
- Types depend on terms;
- Types depend on types

Each option supports different programming paradigms, and relates via Curry-Howard to different notions of logic.

The Lambda Cube



Indeed these three approaches to variables can be combined in various ways.

We will start with λ2 a.k.a. **System F**

- For the programmer, **parametric polymorphism**
- For the logician, **second-order propositional logic**

Diagram c/o joomy on Twitter

Types and Terms of System F

Given a set of **type variables** X,Y,…

- Type variables are types
- If A and B are types then so is A→B
- If A is a type then so is ΠX.A. Note that Π binds its variable

Hence extend STLC with new term-formers, and typing rules:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \Lambda X.t : \Pi X.A} \qquad \frac{\Gamma \vdash t : \Pi X.A}{\Gamma \vdash tB : A[B/X]}$$

The Λ rule is only permitted if X does not appear as a free variable in any type in Γ.

An Example

$\Pi X.X \to X$ is a closed type of `functions $X \to X$ for any X'

So it should be inhabited by a **polymorphic identity function**

- Clearly superior to individually defining functions for each type

$$\dfrac{\dfrac{\dfrac{}{x:X \vdash x:X}}{\vdash \lambda x^X.x:X \to X}}{\vdash \Lambda X.\lambda x^X.x:\Pi X.X \to X}$$

To turn it into an identity function that can be used at some specific type A we write

$$\vdash (\Lambda X.\lambda x^X.x)A:A \to A$$

Reduction for System F

Along with the usual beta-reduction we have

- $(\Lambda X.t)A \mapsto t[A/X]$

The substitution will only effect the type annotations in terms

- Which we sometimes omit
- e.g. $(\Lambda X.\lambda x^X.x)A \mapsto \lambda x^A.x$

All desirable properties proved for our weaker systems continue to hold

Note that real polymorphic languages e.g. Haskell tend to use type variables but often omit Λ

- Λ always implicitly at the outermost level

Encodings in System F

Chapter 11 of ‘Proofs and Types’ show that System F is powerful enough to encode many useful types

- Booleans, Products, Sums, Empty type;
- Existential Type (of which more soon);
- Natural numbers, Lists, Trees, inductively defined types more generally…

But polymorphism is not the only way to get access to the ability to define one’s own types

- The most natural approach is to develop STLC in another direction: ‘Types depend on types’
- In this setting e.g. Lists depend on their base type

Curry-Howard for System F

System F has an operator that binds variables at the type level

- As do all the extensions in the lambda cube, in fact

Via Curry-Howard, we would like to relate this to connectives that bind variables in propositions.

We indeed know two such connectives: ∀ and ∃

- So all the extensions in the lambda cube relate to some sort of quantifier:

Terms depend on types	Second order quantification
Types depend on terms	First order quantification
Types depend on types	Higher order quantification

Second Order ∀

Second order (propositional) logic has formulae ∀X.A where X is understood to be an **arbitrary proposition**

- *Not* an arbitrary element of some domain of elements – that is *first* order

So e.g. ∀X.X→X is a theorem

- 'For any proposition X we have X→X'

∀X.X∨¬X is not a theorem, but nor is it a contradiction

- We are still working in intuitionistic logic, albeit in powerful extensions!
- We could take this as an assumption to prove certain things

What About Second Order ∃?

Is there an analogue of second order ∃ that is useful for programming?

Yes!

- Existential types correspond to **abstract data types**.
- Symbol usually written Σ
- By asserting a type exists, but not committing to what it is, we can work (from the outside) with something without caring about its internal representation.

Example: heterogenous lists

- Usual polymorphic lists can only be instantiated to contain data of one type;
- But (assuming we have a polymorphic list constructor) `List(ΣX.X)` is a list of values that each have some type; each type might be different!

Quantifiers in Intuitionistic Logic

Stepping back to the BHK Interpretation:

- A proof of $\forall x:?.A$ is a construction that transforms a proof of $a \in ?$ into a proof of $A[a/x]$
- A proof of $\exists x:?.A$ is given by providing $a \in ?$, and a proof of $A[a/x]$

The ? depends on which sort of logic we are doing

- Left out when I want to be vague, or it is clear from context
- Second order: ? is the set of propositions

So proofs of ∀ propositions are functions, and of ∃ propositions are pairs

- E.g. $\Sigma X.X$ is a pair of a type, and an element of that type
- Programming: usually written $\Pi x:?.A$ and $\Sigma x:?.A$
- But sometimes $(x:?) \to A$ and $(x:?) \times A$

Constructive ∃

Defining ∃ via pairs creates another difference with classical logic.

- $\exists x{:}A \leftrightarrow \neg(\forall x.\neg A)$ and $\forall x.A \leftrightarrow \neg(\exists x{:}\neg A)$ fail
- Analogous to failure of e.g. $A \wedge B \leftrightarrow \neg(\neg A \vee \neg B)$

The impossibility of a certain construction *failing* to exist does not intuitionistically imply that the construction *does* exist.

- We need to provide the element a, *then* prove it has property $A[a/x]$
- To prove an irrational number exists, it is *not* enough to prove that not all real numbers can be rational; an example of an irrational number must be given.
- This is the heart of **constructive proof**.
- A constructive proof of an existential can be used as an algorithm: it tells you how to generate the element that has the desired property.

Dependent Types

Types depend on terms

- Corresponds to **typed** first order logic
- We do not merely write $\forall x.B$, but instead $\forall x:A.B$
- So the elements used to instantiate variables are proofs / programs.

The proof / typing rules resemble those of propositional logic:

$$\frac{\Gamma, x:A \vdash t:B}{\Gamma \vdash \lambda x^A.t:\Pi x:A.B} \qquad \frac{\Gamma \vdash t: \Pi x:A.B \quad \Gamma \vdash u:A}{\Gamma \vdash tu:B[u/x]}$$

Dependent Types for Theorem Checking

Dependent types are the central technology behind many (not all!) **interactive theorem provers / proof assistants**

- Coq, Lean, Agda, Idris, etc…
- Dependent types key, but actually at top right of the lambda cube
- The most spectacular and useful application of Curry-Howard
- Types can express propositions (e.g. mathematically meaningful statements), and programs are proofs of them
- If a program **type-checks** (done automatically) then it is correct!
- **Machine-checked** (not usually machine-generated) mathematics.

Some Special Dependent Types

Predicate logic often includes **identity** / equality as a connective

- In dependent type theory we have for any type `A`, and any two terms `t` and `u` of that type, a type $\texttt{t} =_{\texttt{A}} \texttt{u}$. Note that this type depends on terms!
- The type $\texttt{t} =_{\texttt{A}} \texttt{t}$ is always inhabited; its introduction is called $\texttt{refl}_{\texttt{t}}$

We usually blur the distinction between types and terms via special types called **universes**, which are collections of types.

- Not all terms are types, but *all types are terms*, i.e. `Bool` is a term of type `Type`, so is `Bool × Nat`, etc.
- More than one universe is necessary because `Type : Type` causes Girard's paradox; as with the similar Russell's paradox, solve with a hierarchy of types

Dependent Types in Action

Let's try to prove a simple mathematical statement:

There is no largest natural number

First, let us rephrase our statement without negation to give a more constructive proof:

- For every natural number n there exists a natural number `m` such that `m > n`
- I.e. we will construct a term of type `Πn:Nat.Σm:Nat.m>n`
- We must give an algorithm that, given any n, constructs this `m` *and* proves it has the property that it is bigger than `m`
- For our first task, we will simply use `suc n`. But how should the second task, the proof, look?

Dependent Types in Action: >

If $\Pi n{:}\mathtt{Nat}.\Sigma m{:}\mathtt{Nat}.m > n$ is to make sense, $m > n$ must be a type

- The collection of proofs that m is greater than n
- *Not* an element of $\mathtt{Bool}$

Assuming we have some machinery in our language to define inductive types:

- $\vdash \_>\_ : \mathtt{Nat} \to \mathtt{Nat} \to \mathtt{Type}$
- $(\mathtt{zero} > n) := 0$
- $(\mathtt{succ}\ m > n) := (m =_{\mathtt{Nat}} n) + (m > n)$
- (this + is ‘or’, not addition)

Dependent Types in Action

Given

- $(\mathtt{succ}\ m{>}n) := (m{=}_{Nat}n){+}(m{>}n)$

we can prove $\mathtt{succ}\ m{>}m$:

- $m{:}Nat \vdash \iota^1\ refl_m : succ\ m{>}m$

And can hence prove our target theorem:

$$\vdash \lambda^{Nat}x.\langle succ\ x, \iota^1\ refl_x\rangle : \Pi n{:}Nat.\Sigma m{:}Nat.m{>}n$$

So the Curry-Howard isomorphism has been used to prove a real statement of mathematics, in a style a computer could check!

Next steps: Exploring the Theory

- Proofs and Types by Girard
- Much more in-depth: Lectures on the Curry-Howard Isomorphism by Sørensen and Urzyczyn
- Follow citations from Propositions as Types by Wadler
- Look at some of my work on links between type theory and *modal* logic: Fitch-Style Modal Lambda-Calculi, or Modal Dependent Type Theory and Dependent Right Adjoints
- Category theory is the inescapable mathematics behind almost all of the mathematics of type theory

Next steps: Curry-Howard in Practice

Get started with Coq, or Lean, or Agda, or Idris!

Multiple members of ANU's Foundations Cluster ready to supervise projects on theorem provers based on Curry-Howard

- As well as experts on theorem provers based on Higher Order Logic

Any More Questions?

And…

Thanks for your time!