DD2552 Seminar 12: Advanced topics

Karl Palmskog

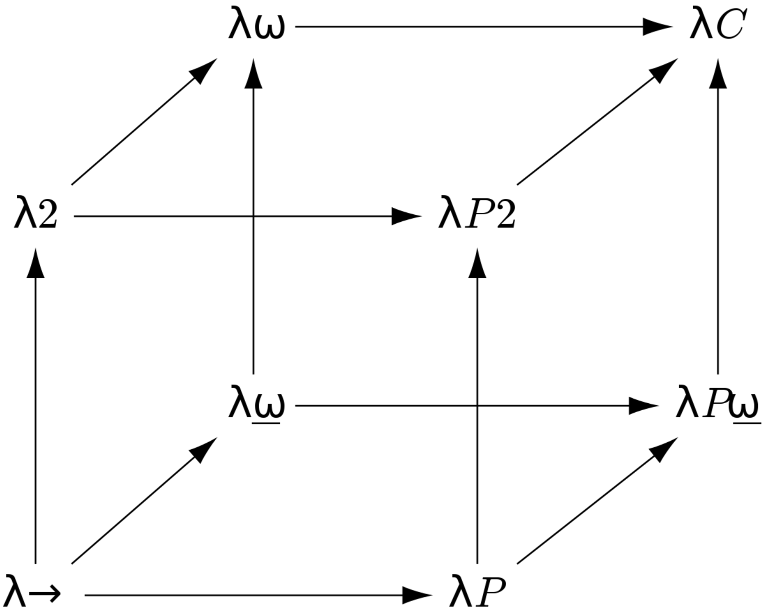
KTH

Wednesday October 11, 2023

Course material

- https://plato.stanford.edu/entries/type-theory-church/
  - Church: A Formulation of the Simple Theory of Types (1940)
- Thorsten Altenkirch: Dependent types, http://www.cs.nott.ac.uk/~psztxa/oplss-22/dependent.pdf
- https://plato.stanford.edu/entries/type-theory-intuitionistic

The lambda cube



Breakdown of lambda cube

- x-axis: types that can bind terms
- y-axis: terms that can bind types
- z-axis: types that can bind types

- $\lambda \rightarrow$, simply typed lambda calculus
- $\lambda 2$, System F, "second order lambda calculus"
- $\lambda \underline{\omega}$, System F$\underline{\omega}$, "types depend on types"
- λP, Lambda-P, "Logical Framework"
- $\lambda \omega$, System Fω, terms/types depend on types
- λC, Calculus of Constructions, terms/types depend on terms/types

Significance of lambda cube

- foundations for different languages and tools
  - $\lambda 2$ (System F) for Haskell, Standard ML
  - λC (Calculus of Constructions) for Coq, Idris
- expressive power vs. decidability and ease of implementation

Functional languages and logic

- lambda cube languages describe computable functions
  - higher-order functions
  - `bool` is a datatype
  - reasoning happens outside language
- first-order logic describes (semi-decidable) formulas
  - reasoning is the whole point
  - encoding of some functions using signatures
- can we combine lambda functions and logic?

Simply typed lambda calculus as a logic

Types:

- ι is the type of individuals
- o is the type of truth values (Booleans)
- if α and β are types, then $\alpha \to \beta$ is a type

Primitive constants:

- $\sim$: $o \to o$ (negation)
- $\vee : o \to (o \to o)$ (disjunction)
- $\prod : (\alpha \to o) \to o$ (for all)
- $\epsilon : (\alpha \to o) \to \alpha$ (choice)

Derived constants:

- $A \wedge B$ is $\sim (\sim A \vee \sim B)$
- $A \Rightarrow B$ is $(\sim A) \vee B$
- $\forall x_\alpha . A_o$ is $\prod(\lambda x_\alpha A_o)$

Equality

- we can quantify over all predicates (one-place functions on o)
- this allows representing equality using the "Leibniz approach"
- we can also represent induction principles

Define Q as

$$\lambda x_\alpha.\lambda y_\alpha.\forall f_{\alpha\to o}.f\,x \Rightarrow f\,y$$

Then define $A_\alpha = B_\alpha$ as $Q\ A_\alpha B_\alpha$. What is the type of Q?

Example

"Napoleon's soldiers admire him"

$$(\lambda n_\iota.\forall x_\iota.\mathrm{Soldier}_{\iota\to o}\, x_\iota \wedge \mathrm{CommanderOf}_{\iota\to\iota} x_\iota = n_\iota$$
$$\Rightarrow \mathrm{Admires}_{\iota\to(\iota\to o)} x_\iota n_\iota)\mathrm{Napoleon}_\iota$$

Axioms in the system

- Alpha-conversion: changing the names of bound variables consistently.
- Beta-contraction: performing λ-application-substitutions inside terms.
- Beta-expansion: infer C from D if D can be inferred from C by one beta-contraction.
- Substitution: from $F_{\alpha \to o}\, x_\alpha$, infer $F\, A_\alpha$ when x not free in F
- Modus Ponens: from $A_o \to B_o$ and A_o, infer B_o
- Generalization: from $F_{\alpha \to o}\, x_\alpha$, infer $\forall x_\alpha . F x_\alpha$ when x not free in F
- Boolean and function extensionality
- Choice: $(\exists x_\alpha . P\, x) \Rightarrow P\, (\epsilon\, P)$
- Axiom of infinity

Dependent types

- not allowing types to depend on terms means no type `R^n` of vector spaces, for `n : nat`
- Calculus of Constructions lifts this restriction
- we assume
  - a universe of types U
  - a type $A : U$
  - a family of types $B : A \to U$
- dependent functions: $(x : A) \to B(x)$, also $\prod(x : A)B(x)$
- dependent sums: $(x : A) \times B(x)$, also $\sum(x : A)B(x)$
- if B does not depend on x, i.e., $B(x) = B$ is constant:
  - we get familiar $A \to B$
  - we get familiar $A \times B$

Logical operations with dependent types

- $A \wedge B$ reduces to $A \times B = (x : A) \times B$
- $A \Rightarrow B$ reduces to $A \rightarrow B = (x : A) \rightarrow B$
- $(\forall x : A)B(x)$ reduces to $(x : A) \rightarrow B(x)$
- $(\exists x : A)B(x)$ reduces to $(x : A) \times B(x)$
- $A \vee B$ reduces to $A + B$
- $\sim A$ reduces to $(x : A) \rightarrow \bot$, for empty type $\bot$

How is this different from classical logic?

Example

$$\forall m : \texttt{nat}.\exists n : \texttt{nat}.m < n \land \texttt{Prime}\, n$$

$$\prod m : \texttt{nat}. \sum n : \texttt{nat}.m < n \times \texttt{Prime}\, n$$

Equality and dependent types

- not as straightforward as for simple types
- judgmental equality: normal forms of terms are identical
- can manually define identity relation directly for a type like `nat` ("propositionally equal" `nats`)
- more promising approach: a general "identity type former"
  - does not provide extensional equality (of functions)
  - distinguishes propositional and judgmental equality
- extensional type theory: type checking becomes undecidable
  - but extensional equality (of functions)
  - conflates propositional and judgmental equality

Why logic inside a type theory?

- no separate metalanguage for reasoning
- basic tooling via implementation of type checker
- establish (near-arbitrary) properties of functions and data
- soundness of reasoning reduces to soundness of type theory (rules)