15-819 Computational Type Theory

15-819 Notes from January 16, 2018

Overview of Comutational Type Theory vs Formal Type Theory
Technical invesigation of λ-calculus with function types (simple type theory)

1. Historical context development of critical ideas
   central idea – especially relational interpretation of types

2. foreshadows later development of higher types (coercion of types)
   master theory of type coercion
   (an investigation of the meaning of type equality
   need both formal and computational aspects
   two notions of variable

3. redPRL implementation of CHiTT
   redprl.org

4. fundamentals of computational type theory (0-dimensional)

5. guarded computational type theory (harper and ?)
   talk about "causality"/"time"

6. higher type theory
   cubical type theory (in computational form)

   a) cubical infrastructure
   b) univalence, inductive types
      identify (treat as equal) eqivalent types

$$\frac{A \leftrightarrow B}{A = B}$$

      if equivalence then equal

7. Computational Type Theory as a specification language for program verification, especially with regard to state, partiality

8. **Types are more than just propositions**
   (Down with Curry-Howard rant) as a formal but uninteresting observation

9. Type theory vs theory of proofs

10. Type theory as a theory of truth (Brouwer/Heyting/Kolomogrov)
    Author's Comment: Heyting, A. "Intuitionism: An Introduction" (1956) North-Holland Publishing ISBN 7204-2239-6

11. Types are parasitic on computation

12. comparison with higher type theory
    - cartesian form (normal cube)
    - kleene form (Kleene algebra cube)
13. separation logic (Reynolds)
    - type theory for concurrency
    - action and session tyhpes (substructural logic)
14. See /rwh/courses/chtt
15. Read Proofs and Refuations
    https://math.berkeley.edu/ kpmann/Lakatos.pdf
16. **Formal Type Theory**
    - structural
    - prescriptive
    - axiomatic
17. **Computational Type Theory**
    http://www.nuprl.org/documents/Constable/NaiveTypeTheoryPreface.html
    - behavioral
    - descriptive
    - semantic
18. Formal Type Theory Formal logic (bunch of rules / derivation tree)

$$\begin{array}{llll} \Gamma & \vdash & A\ type & \qquad \Gamma \vdash A \equiv B \\ \Gamma & \vdash & M : A & \qquad \Gamma \vdash M \stackrel{\leq}{=} N : A \end{array}$$

    (defunitional equality)

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A \equiv B}{\Gamma \vdash M : B}$$

19. Formal Type Theory Methodology
    (a) correspondence to formal logic
        - $\Gamma \vdash A : Prop \approx \Gamma \vdash A\ Type$
        - Prop is Type but (nat, list are not Prop)
        - $\Gamma \vdash A\ true \approx \Gamma \vdash M : A\ (\exists M)$

    (b) constructive (two senses)
        A) computational interpretation
        proofs should run as programs

B) absense of law of excluded middle
axiomatic freedom
- law of excluded middle $\leftrightarrow$ no univalence

(c) decidability
- recursively enumerable $\leftrightarrow$ computationally enumerable
- insist on decidable
- either it is derivable or not
- type checking is proof checking
- issue of complexity (size of proof terms)

(d) Computational meaning is imposed after the fact

20. **Computational Type Theory**

- type theory is **all** about computation
- type theory is foundational in that it is a self-standing theory of truth (don't need sets)
- truth is based on computation because that is the fundamental human faculty (we all understand "algorithm")
Author's Comment: Brouwer's philosophy is that mathematics is fundamentally about human communication and that everyone has an intuitive understanding of an algorithm.
- start with a programming language
  - $M \to M'$ deterministic transition
  - $M \Downarrow V$ (M evaluates to V)
- defines types and elements using "a meaning explanation" (semantics in terms of computation)
- types are certain programs when evaluated their values designate a specification
- "nat" specifies programs that evaluate to O or to the successor
- "if true then nat else 17" is a type
  - elements of a type (obey/satisfy) the specification of the type
  - "if true then 0 else false" *in* nat
  - "if true then 0 else false" *in* "if true then nat else 17"
- meaning of variables in computational type theory is different from variables in formal type theory

$$\begin{array}{ll} \Gamma \gg M \in A & \Gamma \gg M = N \in A \\ \Gamma \gg A \; type & \Gamma \gg A = B \end{array}$$

- no sense checkable
abs $\equiv$ identity in nat
abs $\not\equiv$ identity in $\mathbb{N}$

- type theory specifying behavior, not structure
- in computational type theory the role of type theory is to access the truth
- A is true and proof extracts the program M
- "Refinement Logic" by Constable and Bates
  http://www.cs.cornell.edu/courses/cs4860/2012a/lec-04.pdf
  http://www.redprl.org/pdfs/designing-the-peoples-refinement-logic.pdf
  http://pdfs.semanticscholar.org/c466/665677a8a82a1e5084574972da1a5d11c2bb.pdf
- Chetan Murthy
  https://awards.acm.og/award-winners/MURTHY_6789112