Introduction to Theoretical Computer Science 2017 Summary

Traiko Dinev <traiko.dinev@gmail.com>

July 1, 2019

NOTE: This partially follows Introduction to Theoretical Computer Science, a course at the University of Edinburgh.
NOTE: Note this "summary" is NOT a reproduction of the course materials nor is it copied from the corresponding courses. It was entirely written and typeset from scratch.
License: Creative Commons public license; See README.md of repository

0.1 Pairing

How do we encode many numbers into one? Easy:

$$z = \langle x, y \rangle_2 = 2^x 3^y$$
$$z = \langle x, \langle y, \dots \rangle \rangle$$

In the many number case we need to know how many number we have. We can encode that in the first number, however. This means registers below can represent anything, since they are unbounded.

1 Register Machines

Register machines are like assembly. We have a **fixed** number of registers $R_0, \dots, R_{m-1}$ and a program P, which has n instructions, $I_0, \dots, I_{n-1}$. Things are executed in order, except for $DECJZ$. We have two instructions:

- $INC(i)$ - increment register i by 1
- $DECJZ(i, j)$ - if $R_i = 0$, then goto I_j, else subtract 1 from R

2 Halting Problem and Decision Problems

A machine is encoded as in section 0.1:

$$\ulcorner M \urcorner = \langle \ulcorner P \urcorner, \ulcorner R \urcorner, \ulcorner C \urcorner \rangle$$
$$\ulcorner P \urcorner = \langle \ulcorner I_0 \urcorner, \dots, \ulcorner I_{n-1} \urcorner \rangle$$
$$\ulcorner R \urcorner = \langle \ulcorner R_0 \urcorner, \dots, \ulcorner R_{n-1} \urcorner \rangle$$
$$\ulcorner INC(i) \urcorner = \langle 0, i \rangle$$
$$\ulcorner DECJZ(i) \urcorner = \langle 1, i, j \rangle$$

We thus encode everything, including instructions, registers and the initial register values R.

2.1 Decision Problem

A decision problem is a boolean query onto a subdomain Q in D - (D, Q). The query is, for an element d, is d in Q or not? I.e. is $d \in Q$. We also say $Q(d) = 0?1$. We know that $d \in D$. For machines, D is the domain of all machines; these are integers, encoded as above.

An **oracle** is a machine that knows the answer to our decision problem above. $ORACLE_Q(0)$ is an instruction which calls the oracle for the problem Q and puts the 1 or 0 in R_0.

A **turing transducer** is a machine that translates a decision problem to another one. It takes in R_0 an instance of (D, Q) and outputs in R_0 a $d' = f(d)$ of (D', Q').

A **mapping reduction** or a **many-to-one reduction** is a turing transducer as above, such that $d \in Q$ if and only if (*iff*) $f(d) \in Q'$.

A **M-reduction** is a turing transducer that, after computing d', calls $ORACLE_{Q'}(0)$ and halts. Quoting the $ITCS$ course, "If you allow the transducer to call the oracle freely, you get a Turing reduction. These are also useful; but they're more powerful, so don't make such fine distinctions of computing power.".

Reductions are the way we translate problems to more solvable ones. A reduction $Red(H, Q)$ translates problem H to problem Q. Quoting ITCS "Suppose Q is decidable. Then given $M \in RM$, feed it to $Red(H, Q)$ to decide $M \in H$. But if Q is decidable, we can replace the oracle, and $Red(H, Q)$ is just an ordinary RM. Hence H is decidable contradiction. So Q must be undecidable".

2.2 Halting Problem

Assume a machine H can take in a machine M in R_0 and tell us if M halts. Then make the machine L, which takes a program M_1 in its R_0 and terminates with 1 if H returns 0 and goes in a loop if H returns 1. Thus L loops if its input halts. Now let's diagonalize, i.e. feed L's code into L. If L loops, then by definition L halts and vice-versa. Hence we can not have a machine M. Note this doesn't stop us from having a machine that tells us if a subclass of program halts. We can't have a universal one, that's all.

Variations of the problem:

- Uniform Halting - does M halt on all inputs
- Looping - does M loop on all inputs
- Uniform Looping

Semi-decidable problems are those for which we can say Yes in finite time. **Halting** is one. Interleaving allows us to do this:

- In a loop, go over machines M. They are integers, so we can do this.
- Each iteration simulate all machines before for one step. If one halts, output it.

In finite time, we can output all machines that halt. Hence we can check if some machine halts, but not if it doesn't. Looping is the opposite.

3 Big-O and little-O

$$f \in O(g) \text{ iff } \exists c, n_0. \forall n > n_0. f(n) \leq cg(n)$$
$$f \in \Omega(g) \text{ iff } \exists c, n_0. \forall n > n_0. f(n) \geq cg(n)$$

For little:

$$f(n) \in o(g(n)). \forall \epsilon > 0. \exists n_0. \forall n > n_0. |f(n)| \leq \epsilon |g(n)|$$

3.1 Polynomial Complexity

- PTime: time $f(n) \in O(n^k)$ for some k
- PTime reduction (D', Q') to (D, Q) in PTime, we say $Q' \leq^P Q$

4 Non-Deterministic RM

Add an instruction $MAYBE(J)$, which either $\begin{cases} does\ nothing \\ jumps\ to\ I_j \end{cases}$. In time n, our machine can go over all branches for all maybe instructions. This is, of course, not realistic, unless quantum, which we don't talk about here. Even then it's not exactly so. Our NP (non-deterministic polynomial) machine can examine $2^{O(n)}$ paths in n time. We also have equivalent definitions via a verifier (i.e. the problem can be checked in polynomial time). Check wikipedia or ITCS tutorials for that definition.

- Q is easier than Q' iff $Q \leq^P Q'$
- NP-hard iff $Q \geq^P Q'$ for all $Q' \in NP$
- NP-complete iff NP-hard and NP

4.1 SAT

Suppose $(D, Q) \in NP$. Then by the Cook-Levin theorem $Q \leq^P SAT$ is NP-complete. Trivia - independently discovered by Cook and Levin and we attribute to both after the fall of the iron curtain.

For a decision problem $d \in D$, design a problem ϕ_d which is satisfied if it describes an execution of a a NRM (non-determnistic RM) checking Q. Size of ϕ_d is polynomial in d. We basically encode everything, check ITCS or wiki. Other problems:

- CLIQUE: Does the graph G have a k-fully connected subset (clique). Same as SAT, done via boolean expressions equivalence (exercise.)
- INDSET: Inverse of clique, independent set
- 3-SAT: version of sat, automatically we get this if we did SAT via turing machines, instead of register machines

4.2 More Complexity

- EXPTIME: harder than P and NP
- 2-ExpTime $\not\geq$ ExpTime
- PSpace $\supseteq$ PTime
- PSpace $\supseteq$ NPTime
- PSpace $\subseteq$ ExpTime

Note Space = # bits in registers.

4.3 Polynomial Hierarchy

$$\Sigma_1^P = NP^P = NP$$
$$\triangle_{n+1}^P = P^{\Sigma_n^P}$$
$$\triangle_0^P = \Sigma_0^P = \Pi_0^P = P$$

5 Lambda Calculus

A different model of computation, proposed by Alonso Church. Variables are $x, y, z, \ldots$ also known as terms. If $x : \ var$, $t : \ term$ then $\lambda x.t$ is also a term. These are effectively functions.

if s and t are terms, then (st) is also a term.

$$\lambda x.s\lambda y.xty \triangleq \lambda x.(s(\lambda y.((xt)y)))$$
$$stu \triangleq (st)u$$

This is how we group stuff together.

If x is free in t', then it's bound in $\lambda x.t'$. We have rules about how to compute stuff. α-conversion: changes the names of variables, β conversion evaluates things:

$(\lambda x.t)(s) \xrightarrow{\beta} t[s/x]$ where x is replaced by s in t
$(\lambda x.fx) \leftrightarrow^{\eta} f$ eta conversion, go back and forth

Call by name:

$$(\lambda z.w)((\lambda x.x)(\lambda x.xx)) \xrightarrow{\beta} w$$
$$+ n_1 n_2 \xrightarrow{\beta} n \text{ add } n_1 + n_2 \text{ into n}$$
$$\text{if } bst \xrightarrow{\beta} \begin{cases} s \text{ if b is true} \\ t \text{ if b is false} \end{cases}$$

Y combinator, does recursion:

$$Y \triangleq \lambda F.(\lambda X.F(XX))(\lambda X.F(XX))$$
$$YG \xrightarrow{\beta} G(YG)$$

6 Typing Rules in Lambda Calculus

Assign type to each term $\lambda x : \alpha . y : \beta$. Then this is a function, of type $\alpha \to \beta$. Three typing rules:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x : \sigma . t : \sigma \to \tau}$$

$$\frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash ts : \tau}$$

The 0^{th} rule is that $\Gamma \vdash c : \tau$ for all constant c of base types τ. Base types are nat, bool, etc.

Here's a sample inference of types, going from below. The expression is $\lambda f . \lambda x . f(fx)$. Assume that the expression has the type $(nat \to nat \to \alpha)$, since it's a lambda abstraction taking an argument of type $(nat \to nat)$ and returning some type α. Our goal is to find a type for α.

$$\dfrac{\dfrac{\dfrac{}{f : nat \to nat, x : nat \vdash f : \gamma \to \beta} \qquad \dfrac{f : nat \to nat, x : nat \vdash f : \epsilon \to \gamma \qquad f : nat \to nat, x : nat \vdash x : \epsilon \qquad [\epsilon = nat]}{f : nat \to nat, x : nat \vdash fx : \gamma \qquad [\gamma = nat, \beta = nat]}}{f : nat \to nat, x : nat \vdash f(fx) : \beta}}{\dfrac{f : nat \to nat \vdash \lambda x : nat . f(fx) : nat \to \beta \qquad [\alpha = nat \to \beta]}{\lambda f : nat \to nat . \lambda x : nat . f(fx) : nat \to nat \to \alpha}}$$

6.1 Recursion and fix

Fix for recursion:

$$\text{fix}(\lambda x : \tau . t) \xrightarrow{\beta} t[\text{fix}(\lambda x : \tau . t)/x]$$

Typing rule for fix:

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash \text{fix } t : \tau}$$

Define **let** and **letrec**, as in Haskell:

- **let** $x : \tau = t$ **in** $t' \triangleq (\lambda x : \tau . t')t$
- **letrec** $x : \tau = t$ **in** $t' \triangleq$ let $x : \tau = \text{fix}(\lambda x : \tau . t)$ **in** t'

Type substitution [nat/ α, ...]; $[\text{nat}/\ \alpha](\lambda x : \alpha . x) = \lambda x : nat . x$. Substitution preserves typing (theorem): If $\Gamma \vdash t : \sigma$, ξ is a substitution, then $\xi\Gamma \vdash \xi t : \xi\tau$.

Polymorphism is only allowed in let:

$$\text{let id} = \lambda x . x \textbf{ in } \langle \text{id}(1), \text{id}(true) \rangle$$

Then **id** is polymorphic. Hindley-Milner defines the type inference rules. **let** $x = t$ **in** t' is a term:

let $x = t$ **in** $t' \xrightarrow{\beta} t'[t/x]$

This has the following typing rule for generalization:

$$\frac{\Gamma \vdash t : \sigma \qquad \Gamma, x : \overline{\Gamma}(\sigma) \vdash t' : \tau}{\Gamma \vdash \textbf{let } x = t \textbf{ in } t' : \tau}$$

and for specialization:

$$\frac{x : \sigma \in \Gamma \qquad \sigma \sqsubseteq \tau \text{ (this is specialization)}}{\Gamma \vdash x : \tau}$$

Finally, the rule for specialization is:

If $x : \alpha \vdash t : \alpha \to \beta$, then $\overline{\{x : \alpha\}}(\alpha \to \beta) = \forall \beta . \alpha \to \beta$

7 References and Links

- `http://www.inf.ed.ac.uk/teaching/courses/itcs/` - Edinburgh course on theoretical computer science. What most of this note is based on.
- Michael Sipser Introduction to the Theory of Computation, PWS Publishing (International Thomson Publishing)
- IBenjamin C. Pierce Types and Programming Languages, MIT Pres