Advanced Progamming Languages **Lecture 16**
CS 6110 Spring 2015 **Fri. Feb. 27, 2015**

Lecture 16

Topics

1. Review of *applied* lambda calculus – sample design issues.
2. Explaining the Kleene Normal Form theorem as a "universal machine", other abstract concepts about computable function formalisms used in mathematics – s-m-n, recursion theorems.
3. Indexing as an abstract approach to programming languages.
4. Chuch-Rosser theorem – first look (See textbook, page 163)

---

1. **Applied Lambda Calculus**

   Dr. Rahli introduced us to applied λ-calculi having numbers, addition, and the *let* operator for call-by-value application. He was making *design choices* for pedagogical purposes. Eventually we will see a "full-fledged" applied lambda calculus when we study constructive type theory.

   His version had numbers, addition and $\underline{\text{let}}\ x = a\ \underline{\text{in}}\ b$, the call by value operator. Later we will add many more kinds of primitive data and operations on it. We know from Kleene that numbers are enough along with elementary functions, and the least number operator.

   Another way to write *call-by-value* is to have an operator that looks like $cbv(f; a)$ to complement $ap(f; a)$. We could also write this as $let(a; x.b)$. These are design choices.

2. **Kleene Normal Form Theorem and its consequences**

   How does the normal form theorem work? Consider

   $$\varphi(x_1, ..., x_n) = U(\mu y.T_n(e, x_1, ..., x_n, y)).$$

   The number e is a *numerical code* for the function φ being defined by general recursion. It arises from "arithmetizing" (or gödelizing) the equations. Kleene does this in great deatail. It is an implementation, taking almost 20 pages in his 550 page book.

   The number y codes up a computation. The T-predicate says in detail what a computation of φ is, step-by-step and checks that the number y really is the code of a computation.

We could code up an abstract machine for a λ-calculus with numbers, a conditional (if $x = y$ then __ else __), and a *fix* operator as a general recursive function. We would need the termination guarantee. We could then treat $\varphi(x_1, ..., x_n)$ as a λ definable function.

We get a version of this Normal Form Theorem also for the *partial recursive functions.* It is written with an equality similar to Howe's computational equality, $\simeq$.

Theorem 19 *$\varphi(x_1, ..., x_n) \simeq U(\mu y.T_n(e, x_1, ..., x_n, y))$ and $\exists y.\, T(e, x_1, ..., x_n, y)$ is the assertion that $\varphi(x_1, ..., x_n)$ halts.*

Kleene's famous *recursion theorem* is this:

$$\{e\}(x_1, ..., x_n) \simeq \psi(e, x_1, ..., x_n).$$

This says that we can "solve" any recursion of the form $z(x_1, ..., x_n) \simeq \psi(z, x_1, ..., x_n)$ for the numerical value z.

3. Abstract approach to programming languages

This numerical indexing idea led Rogers[4] to a very elegant and general account of recursive function theory. He defined an *acceptable indexing* of the partial recursive functions as a mapping $\varphi : \mathbb{N} \to \mathbb{PR}$ satisfying exactly two conditions.

1. There is a *universal machine* in φ, i.e. for any pairing function (computable) $p : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ that is 1-1 onto, there is a $m \in \mathbb{N}$ such that $\varphi_m(p(x, y)) \simeq \varphi_x(y)$
2. There is a total recursive function s such that $\varphi_{s(i,m)}(n) \simeq \varphi_i(m, n)$ for all i, m, n. (This is called the s-m-n condition.)

In the abstract approach to programming, it is possible to define computational complexity measures in a very general way. Here is how Manuel Blum did it in his 1967 Journal of the ACM article.[2]

Associate with every φ_i a *complexity function* Φ_i such that

B1. $\varphi_i(x) \downarrow$ iff $\Phi_i(x) \downarrow$ and

B2. Require a total recursive function $M(i, n, m)$ such that

$$M(i, n, m) = \begin{cases} 1 & \text{if } \Phi_i(n) = m \\ 0 & \text{otherwise} \end{cases}$$

It is possible to show that there is a function β on indices such that $\Phi_i = \varphi_{\beta(i)}$.

Blum's best known theorem from this paper is called the *speed-up theorem.* It says that there are computable functions with no best (or even good) algorithms.

Blum Speed-up Theorem *Given any general recursive function $r : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, we can find a 0, 1 valued recursive function f such that for every index i for f there is another index j for f such that $\Phi_i(n) > r(n, \Phi_j(n))$ for almost all n.*

Other well known results in this subject can be found in the survey article by Hartmanis and Hopcroft from 1971[3]. The well known Borodin Gap Theorem is covered in this article. Allan Borodin is basically the first PhD graduate from the Cornell Computer Science department. His result was recently formalized[1]. Later in the course we will examine how computational complexity can be expressed in type theory, and we will mention the famous results on probabilistically checkable proofs (PCP).

References

[1] Andrea Asperti. A formal proof of Borodin-Trakhtenbrot's Gap theorem. In *Third International Conference, CPP 2013*, pages 163–177. Melbourne, Australia, December 2013.

[2] M. Blum. A machine independent theory of computational complexity. *Journal of the Association for Computing Machinery*, 14:322–336, 1967.

[3] J. Hartmanis and J. E. Hopcroft. An overview of the theory of computational complexity. *Journal of the Association for Computing Machinery*, 18(3):444–475, July 1971.

[4] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill Book Company, New York, 1967.