Lecture 17

Topics

1. Church Rosser Theorem.
2. Typed λ-calculus – We've already seen this in the study of evaluators, we'll show something very neat about types and typed λ in next lecture.
3. Review evaluators.
   - Substitution – Lecture 2 definition, Lecture 4 evaluation function
   - Environments
     Dynamic scoping
     Static scoping and closures (closure convention)
   - CPS style.
   - Suggest a good exam question and bring it to me in class on Wednesday.
4. Reflect on types, compare CPS style to Kleene Normal Form.
5. Other key topics for the midterm exam:

   Barendregt variable convention
   Equational λ calculus theory
   Howe's equality – what it means, not the proofs
   Structural induction on λ-terms
   Subrecursive languages – primitive recursion, CoqPL
   Kleene normal form, universal machines
      μ-operator
      Partial recursive vs. total recursive
      Kleene equality $t \simeq t'$

---

1. **Church-Rosser Theorem** (Thompson p.37)

$$\begin{tikzcd}
& e \arrow[dl] \arrow[dr] & \\
f \arrow[dr] & & g \arrow[dl] \\
& h &
\end{tikzcd}$$

If e reduces to f and e reduces to g using a sequence of β-reductions, then we can find a term h such that $f \to h$ and $g \to h$.

This is not so key when a programming language dictates one strategy, e.g. lazy, or when additional notations indicate the method of reduction, e.g $ap(f;a)$ vs. $cbv(f;a)$.

2. **Typed λ-Calculus** (Thompson section 2.6, p.42)

   Base types, e.g. $\mathbb{N}$.

   Function types $\alpha \to \beta$. (Thompson writes $\alpha \Rightarrow \beta$)

   The type of functions that accept inputs of type α and produce outputs of type β.

   Two styles:
   Curry– types not required on the terms (Nuprl), e.g. $\lambda(x.x) \in \alpha \to \alpha$.
   Church – types attached to terms (Coq), e.g. $\lambda(x^\beta.\lambda(y^\alpha.x)) \in \beta \to (\alpha \to \beta)$.

   Theorem *In the typed λ-calculus, every reduction sequence terminates.*

   We will discuss constructive vs. non-constructive proofs of this theorem. The result holds even if the types are partial types.

3. **Review evaluators and typing**

   0. Basic Types

      Term is our recursive definition of λ-terms

      $$\begin{array}{lcll} \text{Term} & = & \text{Var} & \\ & & \mid \lambda(v.t) & v \in Var,\ t \in \text{Term} \\ & & \mid ap(f;a) & f,\ a \in \text{Term} \end{array}$$

      The values are *closed abstractions*, $\lambda(v.t)$, i.e. no free variables in t.

   1. Substitution Evaluator[1] $eval_0 : Term \to Term$
      (lazy, call-by-name)

      $$\begin{array}{lcl} eval_0(x) & = & x \\ eval_0(\underline{\lambda}(x.b)) & = & \underline{\lambda}(x.b) \\ eval_0(\underline{ap}(f;a)) & = & \text{let } \underline{\lambda}(x.b) = eval_0(f) \\ & & \text{in } eval_0(b[a/x]) \end{array}$$

      Note: f, b, a are *syntax*

      In the typed λ-calculus of Thompson 2.6, the functions do not include recursive definitions, just the *simply typed λ-calculus.* A key result we will prove is that all reduction sequences terminate.

[1]The actual type is $Term \to \overline{Term}$, where $\overline{Term}$ is either a diverging term or a regular value.

2. Evaluation with environments $\quad eval_d : (Term \times (Var \to Term^+)) \to Term^+$

   (a) Simple Environments - *dynamic scope*

   Let $Term^+$ be $Term \cup \{error\}$, then environments are $Var \to Term^+$.

$$\begin{array}{lcl} eval_d(x,e) & = & e(x) \qquad \text{returns term or error} \\ eval_d(\underline{\lambda}(x.b),e) & = & \underline{\lambda}(x.b) \\ eval_d(\underline{ap}(f;a),e) & = & \text{let } \underline{\lambda}(x.b) = eval_d(f,e) \\ & & \text{in } eval_d(b, e[x \to eval_d(a,e)]) \end{array}$$

   (b) Environments with *closures – static scope* $\quad Env = Var \to (Term \times Env)$
   (recursive type definition)

$$\begin{array}{lcl} eval_c(x,e) & = & \text{let} < t, e' >= e(x) \\ & & \text{in } eval_c(t,e') \\ eval_c(\underline{\lambda}(x.b),e) & = & < \underline{\lambda}(x.b), e > \\ eval_c(\underline{ap}(f;a),e) & = & \text{let} < \underline{\lambda}(x.b), e' > = eval_c(f,e) \\ & & \text{in } eval_c(b, e'[x \to < a, e >]) \\ eval_c : (Term \times Env) & \to & (Value \times Env) \end{array}$$

3. Continuation Passing Evaluator (CP Style - CPS)

$$\begin{array}{lcl} eval_{cp}(x,e,k) & = & \text{let} < t, e' >= e(x) \\ & & \text{in } eval(t,e',k) \\ eval_{cp}(\underline{\lambda}(x.b),e,k) & = & k(< \underline{\lambda}(x.b), e >) \\ eval_{cp}(\underline{ap}(f;a),e,k) & = & eval_{cp}(f,e,k') \end{array}$$

   Where $k' = \lambda(p.eval_{cp}(p.1, p.2[x \mapsto < a, e >], k))$, for p the pair of a function $p.1$, and an environment, $p.2$.

   We use the notation $p.1$ and $p.2$ to pick out the first and second elements of the pair of a function with its environment. So if p has the value $< \underline{\lambda}(x.t), e >$ then $p.1 = \underline{\lambda}(x.t)$ and $p.2 = e$.