Advanced Progamming Languages	**Lecture 25**
CS 6110 Spring 2015	**Mon. March 23, 2015**

Lecture 25

Topics

1. Overview.
2. λ-decidability is limited by halting problem argument.
3. Unsolvability can be expressed using partial types.
4. Elements of Basic Recursive Function Theory (BRFT) in type theory with partial types.

---

1. **Overview**

   We have briefly studied some elements of a theory of partial recursive functions over the type $\overline{\mathbb{N}} \to \overline{\mathbb{N}}$. This led us to have an intuitive and informal account of partial functions on $\mathbb{N}$ and "partial objects" more generally. We formulated Kleene's Recursion Theorem in this setting and looked at two ways to prove it.

   (i) Highly general account using well orderings on sets with special elements $\perp$ representing diverging elements and monotone functions with respect to $a \sqsubseteq b$ on these *ordered sets*.

   (ii) *Computational accounts*[1]

      (a) Kleene's proof of his recursion theorem.

      (b) The Manna account of domains with a bottom element, the $\sqsubseteq$ order relation, and continuous function.

   Optional exercise: Do Kleene "computations" for ASMs. Study his proof.

   We saw how Edinburgh LCF formalized an account of general domains, centered on fixed-point induction. We note two issues with this principle.

   (a) The admissibility criteria for *fixed-point induction* are weak.

   (b) The extensions of arithmetic operations from $\mathbb{N}$ to $\overline{\mathbb{N}}$ is not entirely obvious. For example, consider $0 * \perp = ?$, $\perp * 0/0 = ?$, $\perp /0 = ?$.

[1] But Kleene's computational account in terms of equations is not as "intuitive" as on ASMs.

When we get to type theory, we will see how the fixed point induction issue was clarified by Cornell PhD students, and how the principles of "domain extension" were clarified in *general theory of partial types.*

In this lecture we explore the topic of partial types further by looking at a computability theory over $\overline{\mathbb{N}}$ developed by Scott Smith and me. Later I will show how Crary extended this theory to clarify fixed point induction and how Abhishek Anand advanced issue (b), the general theory of domain extension and *computational domain theory.*

Crary: A type is *admissible* (for induction) iff $f \in (T \to \overline{T}) \Rightarrow fix(f) \in \overline{T}$

Fact: There exist inadmissible types. Optional exercise – find one.

Least Upper Bound Theorem (Crary 1998) *For all f, t, e (where f is closed),* $\forall j : \mathbb{N}.e(f^j/x) \leq t \Rightarrow e(fix(f)/x) \leq t.$

This allows Crary to identify a large class of admissible types. He built on Howe's equality. The progression was:

Park 1970 Machine Intelligence	Milner/Park et al, fixed point induction
Scott 1972	Smith 1989
Igarashi 1972	Smith 1989
The Semantics of Pascal in LCF, L.Aiello, M. Aiello, R. W. Weyrauch, Stanford, 1974	Anand

Crary Definition:

(a) A type A is *admissible* if the upper bound $t(fix(f))$ of an approximation chain $t(f^0), t(f^1), ..., f(f^n), ...$ belongs to A whenever a cofinite subsequence of the chain belongs to A.

(b) A type T is *predicate-admissible*, $T[e^{[k]}] \Rightarrow T[e^{[\omega]}]$ for all k greater than some j. This is the idea from domain theory, like Igarashi 1972.

2. Our λ-calculus theory gives us a weak form of the Halting Problem. This is explained in computer science where we introduce unsolvability and elements of recursive function theory in terms of a single theory of partial types.

Computational Foundations of Baasic Recursive Function Theory, Theoretical Computer Science, v.121, 1993, 89-112.

Theorem *There is no λ-definable total function to decide halting in the untyped λ-calculus (p.91).*

Proof: Suppose h existed, say $h(x) = 1$ if $x \downarrow$ and $h(x) = 0$ if $x \uparrow$. Define $d = Y(\lambda x.h(x) = 1 \text{ then } \Omega \text{ else } 0)$. Note $Y(d) = d(y(d))$, and $d = h(d) = 1$ then $\uparrow$ else 0.

Now consider how d executes.

If $h(d) = 1$ then d diverges, contradicting the definition of h. So we must have $h(d) = 0$. But in this case, the behavior also contradicts the definition of h.

Hence, there is no such h.

Note, the reasoning is constructive, we do not say $(h(d) = 0) \vee (h(d) = 1)$ based on logic, but on the definition of h.

■

3. Constructive PCF (or $\overline{\text{PCF}}$ or $\overline{\text{LCF}}$).

The article is posted with this lecture, referring to section 2 of the paper:

2. A theory of computing (ATC) (A Theory of Partial Computing – TPC)

2.1 Nature of the theory

Do not adopt Church's Thesis.
Use types T and partial types $\overline{T}$

(Would not necessarily talk about computations as objects, just use computation rules that can generate unbounded reductions.)

2.2 Possible interpretations.

2.3 Syntax – terms

Numbers 0,1,2...

Abstraction $\lambda x.t$

Application $f(a)$, we use $s(t)$

Sequentialization $s; t$ do s, then t

succ(r) – successor, $S(n)$

pred(r) – predecessor, $pred(n)$, $pred(0)$ is 0

$zero(r; s; t)$

$fix(f)$

Syntax – types

$\mathbb{N}$	$(S \to T)$
1 (Unit)	$\overline{S}$ (Where S is a bar type)
2 (Bool)	

2.4 The theory

The meaning of types – need to define T before $\overline{T}$, $s = t \in T$ and $t \in T$ as $t = t \in T$. We use $t = t'$ in $\overline{T}$ rather than $t \simeq t'$ in T.
$t\downarrow$ for t converges and $t\uparrow$ for t diverges.

Axioms

λ-intro

λ-elim (application)

Partial equality (bar intro)

Bar elim, $\quad t \in \overline{T}$ and $t \downarrow$ then $t \in T$