

Introduction to Programming Language Semantics

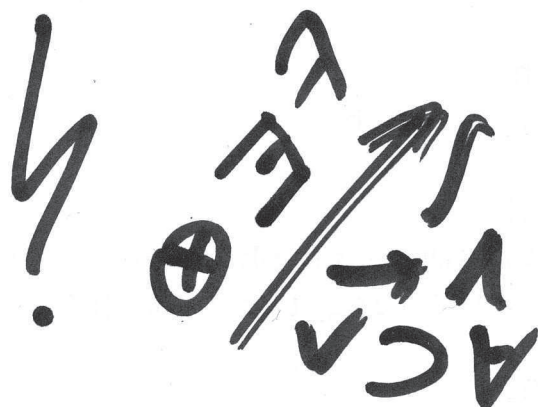


Ohad Kammar

1st Joint Category Theory &
Computer Science Seminar

Cambridge Sunday 18th Nov
2012

Warning:



Careful! Loose
math ahead!

Goals

The audience ~~will know~~ will know the various application domains of CT in semantics.

Subgoals • The audience will know the ~~various~~ use of CT in shaping denotational semantics.

• ~~The audience will know the use of CT in domain theory.~~

• The audience will know the use of CT in programming language structures.

• ~~The audience will know the use of CT in ...~~

PL's → "easy" part, syntax. (Not really easy, but well-studied and formal, so it is into concrete use)

Example syntax

~~size~~

$M ::= \alpha / \text{true} \mid \text{false} \mid n \mid + \mid \times \mid \mid \mid \text{if } M \text{ then } M' \text{ else } M''$
 $\langle M, M' \rangle$ mem/parent
 eg 64-bit address such as 0x3EEF

$C ::= C$
 $C; C$
 $\lambda x. C$

$M ::= M \mid M; M' \mid \lambda x. M \mid M(M')$

What does it mean?

① - fundamental way: show how it computes!

② write a compiler/interpreter. Meaning = how machine behaves.

Problems:

The meeting becomes gcc running on Ubuntu 11.04 on a 32 bit x86 intel machine ... on Wednesday 14th October ... to November...

(not good at all).

Complete implementations are complicated - we want to understand.

Instead: ~~structured~~
~~operational semantics~~

$$\langle \text{Program}, \text{Conf} \rangle \longrightarrow \langle \text{Program}, \text{Conf} \rangle$$

More Formally: define a relation $\longrightarrow$ over configurations $\times$ programs inductively over the syntax:

$$\langle n_1, n_2, \overset{C}{\text{Conf}} \rangle \longrightarrow \langle 7, C \rangle$$

$$\begin{array}{l} \langle \text{if true then } n_1 \text{ else } n_2, C \rangle \longrightarrow \langle n_1, C \rangle \\ \langle \text{if false then } n_1 \text{ else } n_2, C \rangle \longrightarrow \langle n_2, C \rangle \\ \text{Conf: } C = \text{heaps} = \text{Loc} \longrightarrow \text{with finite support.} \end{array}$$

$$\langle M, C \rangle$$

$$\langle l := n, C \rangle \longrightarrow \langle n, C[l \mapsto n] \rangle \quad \langle l!, C \rangle \longrightarrow \langle C(l), C \rangle$$

$$\langle M, C \rangle \longrightarrow \langle M', C' \rangle$$

$$\langle l := M, C \rangle \longrightarrow \langle l := M', C' \rangle$$

(By now the de facto method in the PL community. Not yet in the industry.)

Soundness and types

Programs can get stuck:

if $\langle \text{true} := 5, c \rangle \rightarrow ???$

(e.g., segmentation fault, program crashes, blue screen (if kernel code))

Introduce type systems:

$\Gamma ::= \text{vars} \rightarrow \text{types}$

$\Gamma \vdash M : A$

types: $A ::= \text{Bool} \mid \text{int} \mid A \rightarrow B$
| Loc

$\Gamma \vdash x : A$ $\Gamma(x) = A$

$\Gamma \vdash M_1, M_2 : \text{int}$

$\Gamma \vdash M_1, M_2 : \text{int}$

$\Gamma \vdash M_1 + M_2 : \text{int}$

$\Gamma \vdash M_1 < M_2 : \text{bool}$

$\Gamma, x:A \vdash M : B$

$\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A$

$\Gamma \vdash \lambda x. M : A \rightarrow B$

$\Gamma \vdash M(N) : B$

etc.

Soundness if $\Gamma \vdash M : A$ and c is total then:

when $A = \text{Bool}$ $\langle M, c \rangle \rightarrow^* \langle \text{true/false}, c' \rangle$

$A = \text{int}$ $\langle M, c \rangle \rightarrow^* \langle n, c' \rangle$

$A = A \rightarrow B$ $\langle M, c \rangle \rightarrow^* \langle \lambda x. M', c' \rangle$

Great! We should just let the operational sensors are we're set.

well, not exactly...

e.g. equisoleme

temp := x; ? x := x XOR y;

$$x := y!; j \quad \equiv \quad y := x! \text{ XOR } y!; j$$
$$y := \text{temp!} \quad \sim \quad x := x! \text{ XOR } y!$$

Naive: $\forall c: \langle LHS, c \rangle \xrightarrow{*} \langle n, c_1 \rangle \in \langle RHS, c \rangle \xrightarrow{*} \langle n, c_2 \rangle$

No...⁽²⁾ LHS uses up an extra memory location.

So add: $\text{temp} = 0$

temp $\rightarrow$ at the end.

Useful:

Useful: for all CE-J

define ~~operation~~ observation on texts:

$$C := -|C+M|/|M+C| \text{ if } C \neq M \text{ else } M$$

"n" c else n

" n " n e l x c

$$|c := m| \cdot m := c / \cdot c_j m \mid m_j c \mid \lambda x. c \mid C(m)$$

$$m(C)$$

extend types: $\Gamma \vdash A \vdash C[-]: B$

observational

/Contextual equilibrium

Contextual equivalence:

$$m_1 \equiv m_2 \text{ for all contexts } C[-] : \text{Bool}^A \text{ s.t. } \text{cont } C : \langle e[M_1], C \rangle \xrightarrow{*} \langle b, C' \rangle \text{ iff } \langle e[M_2], C \rangle \xrightarrow{*} \langle b, C' \rangle$$

How to prove $LH R_1 \equiv LH R_2$? Involves some kind of induction over all possible interactions.

very sensitive to language changes, e.g. if we add the ability to run in parallel.

```
C[-] = Parallelize (Temp := 1, -);
```

CE $\rightarrow$ Parallelize (temp := 1, -);

meaning of each program phrase depends heavily on other phrases

Denotational approach

Assign meaning to each term:

easy: $\llbracket \text{true} \rrbracket := 1 \quad \llbracket \text{false} \rrbracket := 0 \quad \llbracket \text{ox BEEF} \rrbracket := 48,879$

compositional: $\llbracket (3+5)*7 \rrbracket := \llbracket 3+5 \rrbracket * \llbracket 7 \rrbracket$

Of course, only for well-typed! : $\llbracket \text{true} := 5 \rrbracket = ???$

So first we define semantics for types; in two

$$\llbracket \text{bool} \rrbracket := 2 \quad \llbracket \text{int} \rrbracket := \mathbb{Z} \quad \llbracket \text{loc} \rrbracket \text{ (e.g. } := 2^{64})$$

$$\llbracket \tau \rrbracket := \prod_{x \in \text{dom} \tau} \llbracket \tau(x) \rrbracket$$

we want

Naïve: $\llbracket A \rightarrow B \rrbracket := \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket \quad \llbracket \Gamma \vdash M : A \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$

and indeed we can interpret:

NA

if M_{bool} then M_{true} else M_{false}

$$\llbracket M \rrbracket(r) := \begin{cases} \llbracket M_{\text{true}} \rrbracket(r) & \text{if } \llbracket M_{\text{bool}} \rrbracket(r) = 1 \\ \llbracket M_{\text{false}} \rrbracket(r) & \text{if } \llbracket M_{\text{bool}} \rrbracket(r) = 0 \end{cases}$$

$$\llbracket M_{\text{bool}} \rrbracket(r) = 1$$

$$\llbracket M_{\text{bool}} \rrbracket(r) = 0$$

but what about $\llbracket M := n \rrbracket(??)$

instead, pass state around: ~~$\llbracket \Gamma \vdash M : A \rrbracket$~~

define $\llbracket \text{Mem} \rrbracket := \llbracket \text{loc} \rrbracket \rightarrow \llbracket \text{int} \rrbracket$ with finite support.

and then $\llbracket \Gamma \vdash M : A \rrbracket \neq \llbracket \Gamma \rrbracket \times \llbracket \text{Mem} \rrbracket \rightarrow \llbracket A \rrbracket \times \llbracket \text{Mem} \rrbracket \cong \llbracket \Gamma \rrbracket \rightarrow (\llbracket A \rrbracket \times \llbracket \text{Mem} \rrbracket)$

Similarly: $\llbracket A \rightarrow B \rrbracket = \llbracket A \rrbracket \rightarrow (\llbracket B \rrbracket \times \llbracket \text{Mem} \rrbracket)$

and now we can give semantics:

$$r \in \llbracket \Gamma \rrbracket, n \in \llbracket \text{Mem} \rrbracket$$

$$\llbracket A \rrbracket(r)(n) = \langle n, n' \rangle \quad \text{if } M_{\text{bool}} \text{ then } M_{\text{true}} \text{ else } M_{\text{false}} \llbracket M \rrbracket(r)(n)$$

$$= \begin{cases} \llbracket M_{\text{true}} \rrbracket(r)(n') \\ \llbracket M_{\text{false}} \rrbracket(r)(n') \end{cases}$$

$$\llbracket M_{\text{bool}} \rrbracket(r)(n) = \langle 1, n' \rangle$$

$$\llbracket M_{\text{bool}} \rrbracket(r)(n) = \langle 0, n' \rangle$$

but now: $\llbracket M_{\text{loc}} := M_{\text{int}} \rrbracket(r, n) := (n, n'[\ell \mapsto n])$

where $\llbracket M_{\text{loc}} \rrbracket(r)(n) = (\ell, n')$

$\llbracket M_{\text{int}} \rrbracket(r)(n') = (n, n')$

and similarly: $\llbracket M_{\text{loc}} ! \rrbracket(r, n) := (n[\ell], n')$

where $\llbracket M_{\text{loc}} \rrbracket(r, n) = (\ell, n')$

Theorem (~~Generalization~~ Soundness of the den. semantics):

if $M \vdash M:A$ and $\langle M, v \rangle \xrightarrow{*} \langle V, v' \rangle$ then $\llbracket M \rrbracket(\nu)(v) = \langle \llbracket V \rrbracket, v' \rangle$

Proof: By induction on terms, but strengthening the hypo:
a ~~value~~ substitution is a ~~function~~ ^{not} σ s.t.

Given $\Gamma \vdash M:A$ and σ ~~subst~~ for all $x \in \text{dom } \Gamma$, $\sigma(x)$ is a value $V: \Gamma(x)$
and $M\sigma = \llbracket M \rrbracket^{\sigma(x)/x}$. define $\llbracket \sigma \rrbracket := \langle \llbracket \sigma(x) \rrbracket \rangle_{x \in \text{dom } \Gamma}$

Then the new hypo is:

if $\Gamma \vdash M:A$, σ a value subst. for Γ and $\langle M\sigma, v \rangle \xrightarrow{*} \langle V, v' \rangle$
then $\llbracket M \rrbracket \llbracket \sigma \rrbracket (\nu) = \langle \llbracket V \rrbracket, v' \rangle$

By induction on $\xrightarrow{*}$

(Corollary: determinacy...)

Theorem: (adequacy) if $\llbracket M \rrbracket = \llbracket M' \rrbracket$ then $M \cong M'$.

Example: (as before)

Proof: Using logical relations ~~define relations:~~

Let $\text{Terms}_A := \{ \vdash M:A \}$ $\text{Values}_A := \{ \vdash V:A \}$

We define relations $R_A^{\text{Val}} \subseteq \llbracket A \rrbracket \times \text{Values}_A / \cong$ $R_A^{\text{Comp}} \subseteq (\llbracket A \rrbracket \times \mathbb{N})^{\mathbb{N}} \times \text{Terms}_A / \cong$

$$R_{\text{bool}}^{\text{val}} := \{ (0, [\text{false}]), (1, [\text{true}]) \}$$

$$R_{\text{int}}^{\text{val}} := \{ (n, [n]) \mid n \in \mathbb{Z} \}$$

$$R_{\text{Lis}}^{\text{val}} = \dots$$

$$R_{A \rightarrow B}^{\text{val}} := \{ (f, [M]) \mid \text{for all } (a, N) \in R_A^{\text{val}}, (f(a), [M(N)]) \in R_B^{\text{comp}} \}$$

$$R_A^{\text{comp}} := \{ (w. (a_n, \vec{N}_n), [M]) \mid \text{for all } n \in \mathbb{N}, \# \langle M, N \rangle \xrightarrow{*} (V, \vec{N}_n) \text{ and } (a_n, [V]) \in R_A^{\text{val}} \}$$

$$R_{\Gamma}^{\text{val}} := \{ \langle \sigma, \sigma \rangle \mid \text{dom } \sigma \text{ is a substitution for } \Gamma \text{ and for all } x \in \text{dom } \sigma, (M(x), \sigma(x)) \in R_{\Gamma(x)}^{\text{val}} \}$$

Basic lemma: if $\Gamma \vdash M : A$ and $(\Gamma, \sigma) \in R_{\Gamma}^{\text{val}}$ then $(\llbracket M \rrbracket(\sigma), [M\sigma]) \in R_A^{\text{comp}}$

Proof: by induction on $\Gamma \vdash M : A$, for example:

for $\Gamma \vdash n : \text{int}$ $\llbracket n \rrbracket = \lambda v. \langle n, v \rangle$ if $(\Gamma, \sigma) \in R_{\Gamma}^{\text{val}}$, and $n \in \mathbb{N}$
 then indeed $\# \langle n, v \rangle \xrightarrow{*} \langle n, v \rangle$ and indeed $(n, [n]) \in R_{\text{int}}^{\text{val}}$.

More interestingly, for $\Gamma \vdash M : A \rightarrow B$ $\Gamma \vdash N : A$.
 $\Gamma \vdash M(N) : B$

take $(\Gamma, \sigma) \in R_{\Gamma}^{\text{val}}$ then $(\llbracket M \rrbracket(\sigma), [M\sigma]) \in R_{A \rightarrow B}^{\text{comp}}$ $(\llbracket N \rrbracket(\sigma), [N\sigma]) \in R_A^{\text{comp}}$

take any $n \in \mathbb{N}$ then $\llbracket M \rrbracket(\sigma)(n) = (f_n, n')$ and we have $[N\sigma]_{n'} \xrightarrow{*} (V_{\text{fun}}, n'')$ and $(f_n, [V_{\text{fun}}]) \in R_{A \rightarrow B}^{\text{comp}}$
 for n' we then have $\llbracket N \rrbracket(\sigma)(n') = (a, n'')$ and we have $[N\sigma]_{n'} \xrightarrow{*} (V_A, n'')$ and $(a, [V_A]) \in R_A^{\text{val}}$

But then ~~$(\llbracket M(N) \rrbracket(\sigma), [M(N)\sigma]) \xrightarrow{*} (V_{\text{fun}}(V_A), n'')$~~
 as we also

As we have $(f_n, [V_{\text{fun}}]) \in R_{A \rightarrow B}^{\text{comp}}$, then for all $(a, [V_A]) \in R_A^{\text{val}}$ we have
 $(f(a), [V_{\text{fun}}(V_A)]) \in R_B^{\text{comp}}$. But then:

for all n : $((M(N))\sigma, n) \xrightarrow{*} (V_{\text{fun}}(N\sigma), n') \xrightarrow{*} (V_{\text{fun}}(V_A), n'')$ and hence
 $V_{\text{fun}} V_A \cong (M(N))\sigma$ hence:

$$(\llbracket M(N) \rrbracket(\sigma), [M(N)\sigma]) = (f_n(a), [V_{\text{fun}}(V_A)]) \in R_B^{\text{comp}}$$

Probably no time, but also:

$$\frac{\Gamma \vdash M_{loc} : loc \quad \Gamma \vdash M_{int} : int}{\Gamma \vdash M_{loc} := M_{int} : int}$$

take $(\sigma, \sigma') \in \mathcal{R}_p^{val}$ and any $n \in \mathbb{N}$.

let then $\llbracket M_{loc} \rrbracket(\sigma)(n) = (l, n')$ by induction, $\langle M_{loc}, \sigma, n \rangle \rightarrow^* \langle l, n' \rangle$

let $\llbracket M_{int} \rrbracket(\sigma)(n') = (n, n'')$ by induction $\langle M_{int}, \sigma, n' \rangle \rightarrow^* \langle n, n'' \rangle$

hence: $\langle (M_{loc} := M_{int}), \sigma, n \rangle \rightarrow^* \langle l := M_{int}, \sigma, n \rangle \rightarrow^* \langle l := n, n'' \rangle \rightarrow^* \langle n, n'' \rrbracket [l \mapsto n]$

hence: recall:

$\llbracket M_{loc} := M_{int} \rrbracket(\sigma)(n) = (n, n'' \rrbracket [l \mapsto n])$ so we indeed have:

$$(\llbracket M \rrbracket(\sigma), \llbracket M_0 \rrbracket) \in \mathcal{R}_{int}^{comp}$$

And now we can prove adequacy:

Assume $\llbracket M \rrbracket = \llbracket M' \rrbracket$ Take any $C[-]$ and $(C[M], n) \rightarrow^* \langle v, n' \rangle$.

By compositionality $\llbracket C[M] \rrbracket = \llbracket C[M'] \rrbracket$ so by the basic lemma

we have: $(\llbracket C[M] \rrbracket(\sigma), \llbracket C[M'] \rrbracket) \in \mathcal{R}_{bool}^{comp}$

By Soundness, $\llbracket C[M] \rrbracket(\sigma)(n) = (\llbracket v \rrbracket(\sigma), n')$ hence; by $\mathcal{R}_{bool}^{comp}$ is def:

$(C[M'], n) \rightarrow^* \langle v, n' \rangle$ and we have adequacy. ■

So now we can capture the "meaning" of programs by just calculating $\llbracket M \rrbracket$, and the adequacy theorem guarantees that sometimes comparing with all the other contexts.

But what would happen if we changed the language? What would change, what would stay the same? This is where CT enters the arena.

For example, we replace "memory accesses" with non-determinism.

we have an operation: ~~and~~ $AK(M, N)$ that magically chooses between doing M and doing N .

Operationally, $AK(M, N) \rightarrow M$ and $AK(M, N) \rightarrow N$ (empty cont's) a relation, and $\rightarrow$ becomes finite

Now $\llbracket r \vdash M : A \rrbracket$ becomes $\llbracket M : \llbracket r \rrbracket \rrbracket \rightarrow P_o^{\text{fin}}(\llbracket A \rrbracket)$ the nonempty-powerset.

and similarly the function space $\llbracket A \rightarrow B \rrbracket := \llbracket A \rrbracket \rightarrow P_o^{\text{fin}}(\llbracket B \rrbracket)$

$\llbracket AK(M, N) \rrbracket(r) :=$

$$\llbracket AK(M, N) \rrbracket(r) := \llbracket M \rrbracket(r) \cup \llbracket N \rrbracket(r)$$

The logical relation then becomes:

$$R_A^{\text{G-P}} := \left\{ (X, \llbracket M \rrbracket) \mid \forall x \in X : M \rightarrow^* V, (a, \llbracket V \rrbracket) \in R_A^{\text{val}} \right\}$$

rather
neat

But of course, all the proofs need to be reiterated! And we have other languages: exceptions, I/O, and combinations of them! 2⁵ = 32 already!

So categorically: To give a model of a language we need:

A category $\mathcal{C}$ which has finite products, the ^{distributive} sum $1+1$, a strong monad $T : \mathcal{C} \rightarrow \mathcal{C}$ and $\mathcal{C}$ has Kleisli exponentials.

and interpretations:

And now: $\llbracket \text{int} \rrbracket, \llbracket \text{loc} \rrbracket, \llbracket \text{IO} \rrbracket, \llbracket \text{E} \rrbracket$

For our semantics and for the effects. e.g.

For example: $\mathcal{C} = \text{Set}$ and: for memory, $TA := (A \times M)^M$ with $\llbracket := \rrbracket$ and $\llbracket ! \rrbracket$ as before

for IO, $TA := P_o^{\text{fin}}(A)$

for exceptions: $TA := A + E$ with $\llbracket \text{raise}_M \rrbracket := \text{inj}_2 \llbracket M \rrbracket$

The rest of the semantics is standard, with; for example:

$$\llbracket \text{Bool} \rrbracket := 1+1 \quad \llbracket r \vdash M : A \rrbracket : \llbracket r \rrbracket \rightarrow T \llbracket A \rrbracket \text{ Kleisli arrows}$$

$$\llbracket r \vdash n : A \rrbracket := \eta_{\llbracket A \rrbracket}^{(n)} \quad \llbracket n ; m \rrbracket : \llbracket r \rrbracket \rightarrow \llbracket M \rrbracket T \llbracket V \rrbracket \rightarrow T \llbracket A \rrbracket$$

Kleisli composition

etc.

objects: types, A , with finite products.

Morphisms: ~~Seq~~ equivalence classes:

$$\prod_{x \in \text{Dolr}} P(x) \rightarrow \prod_{i \in I} A_i$$

with values as no $\langle [V_i] \rangle_{i \in I}$ with
 $\hat{\Gamma} \vdash V_i : A_i$. ~~Use $\hat{\Gamma} \vdash V_i : A_i$~~

Composition is given by substitution.

Monad: $TA := (1 \rightarrow A)$

$$\eta_A := a : A \mapsto \lambda x, a : 1 \rightarrow A$$

$$\mu := m : 1 \rightarrow (1 \rightarrow A) \mapsto \lambda x. m(x)(x) : 1 \rightarrow A$$

$$\text{Str}: a : A, m : 1 \rightarrow B \mapsto \lambda x. (\lambda b. \langle a, b \rangle m(x)) : 1 \rightarrow A \times B$$

reasonable operational semantics will move this into a monad,
 and we will have a model Syn .

We then construct another model:

$$\begin{array}{ccc} \mathcal{L} & \xrightarrow{\quad} & \text{Pred} \\ \downarrow & \lrcorner & \downarrow \pi_2 \\ \text{Set} \times \text{Syn} & \xrightarrow{x \text{Syn}(1, -)} & \text{Set} \end{array}$$

$\mathcal{L}$ is the category of logical relations, and we can construct a model in $\mathcal{L}$
 by specifying: $\llbracket \text{int} \rrbracket := \{ \llbracket \text{in} \rrbracket, \llbracket \text{n} \rrbracket \}$ $\llbracket \text{loc} \rrbracket = \{ \llbracket \text{d} \rrbracket, \llbracket \text{r} \rrbracket \}$

and lifting the monad structure T to some $R_A^{\text{Comp}} \subseteq TA \times A_{\text{Terms}} A$

If we give such a model we automatically get a model $\mathcal{L}$, and it being a model is precisely
 a restatement of the basic lemma.

The fact that $\mathcal{L}$ has all the ^{other} required structure follows from fibration abstract

nonseuse (e.g., π_2 being a bi-fibration or for Pred arising out of a factorisation
 system of Set)

And the story continues... ~~structure~~

For some languages, $C = \text{set}$ is not enough, and we have to choose a suitable ~~category~~ different category.

Stephen will talk about ~~a number of~~ categories of domains which we need in order to model recursion (while loops, for example) and algebraic datatypes.

If we want to model locality (local memory for example) we need to switch to ~~first~~ functor categories $C = \text{Set}^I$, or ~~at length~~ to the category of Nominal Sets.

If we want to model (the) concurrency, we need ~~over~~ a category of event structures etc. (domains and event structures also merit CT inside themselves for other reasons...)

The story is far from over...

for example full abstraction & game semantics.

Another key idea that we must mention is new languages.

Semantics are used to reason about our programs. By minimising the semantic gap, we can make our programs easier to understand.

This is the idea behind monads in a PL. Dominic will talk about that, hopefully.

To summarise:

we've looked at operational & denotational semantics, covering type soundness, denotational soundness, adequacy, logical relations.

we've seen how CT helps the Meta-theory by giving a uniform ~~sem~~ action of semantic structure.

~~the structure~~

Recommended reading

Probably covers all topics:

- Winskel, G. (1993). The formal semantics of programming languages. MIT Press.

Operational semantics

- Classical reference on structural operational semantics: (~1981 iirc) Gordon Plotkin: A Structural Approach to Operational Semantics

<http://homepages.inf.ed.ac.uk/gdp/publications/SOS.ps>

Perhaps more contemporary * Cambridge CS Tripos course, e.g.:

<http://www.cl.cam.ac.uk/teaching/1112/Semantics/>

And the recommended reading:

- Pierce, B.C. (2002). Types and programming languages. MIT Press.
- Hennessy, M. (1990). The semantics of programming languages. Wiley. Out of print, but available on the web at

<http://www.scss.tcd.ie/Matthew.Hennessy/slexternal/reading.php>

Denotational semantics

See Winskel's book above, but also the CS Tripos course "Denotational Semantics"

<http://www.cl.cam.ac.uk/teaching/1112/DenotSem/>

The recommended reading from it:

- Winskel, G. (1993). The formal semantics of programming languages: an introduction. MIT Press.
- Gunter, C. (1992). Semantics of programming languages: structures and techniques. MIT Press.
- Tennent, R. (1991). Semantics of programming languages. Prentice Hall.

Categorical models of computation

I'm not sure what's the best place to read about categorical models. You can try the Part III course here:

<http://www.cl.cam.ac.uk/teaching/1213/L24/>

And chase the recommended reading...

It runs on Lent term, so perhaps you want to attend?