Tutorial on Semantics Part I

Basic Concepts

Prakash Panangaden[1]

[1]School of Computer Science
McGill University
on sabbatical leave at
Department of Computer Science
Oxford University

Fields Institute, Toronto: 20th June 2011

Outline

What am I trying to do?

1. Describe the aims and goals of semantics broadly construed
2. Select some few topics to discuss
3. Give some of the main results
4. Give at least a *few* proofs in some detail
5. List things that I have not done

What I am not trying to do

- Cover *all* topics in the field
- Get you to the point where you will follow all the papers in the upcoming conference
- Try to persuade you to work in this area
- Convince you to attend every talk
- Show you how to use category theory to settle the major problems of complexity theory.

What is semantics?

- The search for a structural understanding of computational phenomena
- Computation = dynamics
- Cliché: The only way to deal with complex systems is to decompose them into manageable pieces and understand the pieces.
- Structure: How does one decompose a system?
- Behaviour: How does one understand the *dynamical* aspects of the pieces?
- How does one *compose* the dynamical description of the pieces into a description of the dynamics of the whole system?

What kinds of systems?

- Sequential computer programs are only one example:
- concurrent systems,
- distributed systems,
- reactive systems,
- probabilistic systems,
- real-time systems,
- hybrid systems,
- quantum systems,
- chemical reactions,
- biological systems,
- economic systems,
- business organizations,
- ....

Why are Turing machines wonderful?

- Turing machines give a compelling formalism to justify the idea that computation involves
- a limited number of types of primitive steps
- and each step involves a *finite* piece of information.
- Turing machines show that the dynamical process of computation can be captured by a static entity: the program.
- It gives a notion of “step” which has proved durable and robust, which makes it an ideal formalism for
- quantifying resource use during computation.
- Fantastic for computability theory and complexity theory.

Why can’t we just use Turing machines?

- Just try writing the code for the Fast Fourier Transform as a Turing machine!
- Programming languages were invented for *people* not for computers!
- Well designed programming languages aim to give people the mechanisms to organize their programs in *meaninful structures*.
- Turing machines do not give a *compositional* handle on program behaviour.
- The use of encoding hides concepts like *type* structure.
- Standard concepts – like “what can be computed” ? – have to be revised even for simple extensions like parallel computing.

What tools can we use instead?

- LOGIC!
- Induction
- Algebra: structure
- Colagebra: behaviour
- Coinduction
- Topology: limiting behaviour
- Fixed-point theory
- Measure theory: probabilistic behaviour
- Metric spaces, analysis, topological lattices
- Any damn thing we can lay our hands on!

Syntax

Arithmetic expressions

$$a ::== 0|1|\ldots|a_1 + a_2|a_1 * a_2|\ldots|X|\ldots$$

Boolean expressions

$$b ::== \mathsf{T}|\mathsf{F}|b_1 \textbf{ and } b_2|\ldots$$

Commands

$$c ::== \textbf{skip}|X := a|c_1; c_2|\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2|\textbf{while } b \textbf{ do } c.$$

Operational semantics I

- State: mapping of variables to values. $\sigma : \mathit{Var} \longrightarrow \mathbb{Z}$.
- Semantics of arithmetic expressions: state-dependent values $(a, \sigma) \longrightarrow n$.
- Semantics of boolean expressions: state-dependent values $(b, \sigma) \longrightarrow$ T$|$F.
- Semantics of commands: state-transformers $(c, \sigma) \longrightarrow \sigma'$.
- State update notation $\sigma[X \mapsto n]$.

Operational semantics II

Rules for arithmetic expressions

$$\frac{}{(n,\sigma)\longrightarrow n} \qquad \frac{}{(X,\sigma)\longrightarrow \sigma(X)} \qquad \frac{(a_1,\sigma)\longrightarrow n_1 \quad (a_2,\sigma)\longrightarrow n_2}{(a_1+a_2,\sigma)\longrightarrow n_1+n_2}$$

Rules for boolean expressions

$$\frac{}{(\mathsf{T},\sigma)\longrightarrow \mathsf{T}} \qquad \frac{}{(\mathsf{F},\sigma)\longrightarrow \mathsf{F}}$$

$$\frac{(be_1,\sigma)\longrightarrow b_1 \quad (be_2,\sigma)\longrightarrow b_2}{(be_1\ \mathbf{and}\ be_2,\sigma)\longrightarrow b_1\wedge b_2}$$

$$\frac{(a_1,\sigma)\longrightarrow n_1 \quad (a_2,\sigma)\longrightarrow n_2}{(a_1=a_2,\sigma)\longrightarrow n_1=n_2}$$

Operational Semantics III

Assignment Statement

$$\frac{(a, \sigma) \longrightarrow n}{(X := a, \sigma) \longrightarrow \sigma[X \mapsto n]}$$

Sequential Composition

$$\frac{(c_1, \sigma) \longrightarrow \sigma' \quad (c_2, \sigma'') \longrightarrow \sigma''}{(c_1; c_2, \sigma) \longrightarrow \sigma''}$$

Conditional

$$\frac{(b, \sigma) \longrightarrow \mathsf{T} \quad (c_1, \sigma) \longrightarrow \sigma'}{(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma) \longrightarrow \sigma'}$$

$$\frac{(b, \sigma) \longrightarrow \mathsf{F} \quad (c_2, \sigma) \longrightarrow \sigma'}{(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma) \longrightarrow \sigma'}$$

All these rules are structural.

Operational semantics IV

While loop: false case

$$\frac{(b,\sigma) \longrightarrow \mathsf{F}}{(\mathbf{while}\ b\ \mathbf{do}\ c, \sigma) \longrightarrow \sigma}$$

While loop: true case

$$\frac{(b,\sigma) \longrightarrow \mathsf{T} \quad (c,\sigma) \longrightarrow \sigma' \quad (\mathbf{while}\ b\ \mathbf{do}\ c, \sigma') \longrightarrow \sigma''}{(\mathbf{while}\ b\ \mathbf{do}\ c, \sigma) \longrightarrow \sigma''}$$

The last rule is not structural. It cannot be used in a structural induction proof.

Towards a compositional semantics for while loops

- Note the following equivalence:

$$\textbf{while } b \textbf{ do } c \equiv \textbf{if } b \textbf{ then } c; (\textbf{while } b \textbf{ do } c) \textbf{ else skip}.$$

- We won't worry about the exact meaning of equivalence for now.
- Let's say that commands stand for state transformation functions.
- Write W for the function corresponding to **while** b **do** c
- and Γ for the *transformer* of state-transformation functions given by

$$\Gamma(\theta) = \textbf{if } b \textbf{ then } c; \theta \textbf{ else skip}.$$

- Of course, this is notationally corrupt.
- Then $W = \Gamma(W)$!
- Can we formalize this using fixed-point theory?

Denotational semantics I

- We will formalize the meaning of commands as *partial* functions from states to states.
- Arithmetic expressions will be functions from states to numbers and
- boolean expressions will be functions from states to booleans.
- Notation:

$$[\![a]\!] : St \longrightarrow \mathbb{Z}; \quad [\![b]\!] : St \longrightarrow \mathbf{Bool}; \quad [\![c]\!] : St \rightharpoonup St.$$

Denotational semantics II

Arithmetic expressions

$$[\![X]\!](\sigma) = \sigma(X), \ldots$$

etc. etc.

Boolean expressions

$$[\![a_1 = a_2]\!](\sigma) = ([\![a_1]\!](\sigma) = [\![a_2]\!](\sigma)), \ldots$$

Commands

Functional notation

$\circ$: functional composition

$(\cdot) \Longmapsto (\cdot); (\cdot)$: definition by cases.

Commands

$$\begin{array}{lcl}
[\![\mathbf{skip}]\!](\sigma) & = & \sigma \\
[\![X := a]\!](\sigma) & = & \sigma[X \mapsto [\![a]\!](\sigma)] \\
[\![c_1 ; c_2]\!] & = & [\![c_2]\!] \circ [\![c_1]\!] \\
[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!](\sigma) & = & [\![b]\!](\sigma) \Longmapsto [\![c_1]\!](\sigma); [\![c_2]\!](\sigma) \\
[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] & = & \mathsf{fix}(\Gamma_{b,c})
\end{array}$$

The definition of Γ

- Type: $\Gamma_{b,c} : (St \rightharpoonup St) \longrightarrow (St \rightharpoonup St)$.
- Γ represents one unwinding of the while loop.
- $$\theta : St \rightharpoonup St; \quad \Gamma_{b,c}(\theta)(\sigma) = [\![b]\!](\sigma) \Longmapsto \theta([\![c]\!](\sigma)); \sigma.$$
- How do we know Γ has a fixed point?
- Maybe it has lots of fixed points.

Kindergarten Domain Theory

- The set of partial functions $St \rightharpoonup St$ has a natural order structure.
- If $f \in S \rightharpoonup St$, define the graph of f as $gr(f) = \{(x, y) | y = f(x)\}$ and the domain of f as $dom(f) = \{x | \exists y\, f(x) = y\}$.
- Then define $f \leq g$ by $gr(f) \subseteq gr(g)$.
- The domain of f is contained in the domain of g and when they are both defined they agree.
- There is a least element denoted $\perp$ with $gr(\perp) = \emptyset$ and given a chain $f_1 \leq f_2 \leq \ldots f_n \leq \ldots$ the function f given by $gr(f) = \bigcup_i gr(f_i)$ is the least upper bound of the chain. Let us call such posets *domains* (for now).
- A function between posets that preserves the order is called monotone.
- A function that preserves the lubs of chains is called *continuous*. Continuous functions are always monotone.

The Basic Fixed-Point Theorem

Fixed point theorem

A continuous function on a domain has a *least* fixed point.

Start from $\perp$ and keep applying f.
Since $\perp$ is the least element, $\perp \leq f(\perp)$, since f is monotone $f(\perp) \leq f(f(\perp))$ so inductively we have the chain

$$\perp \leq f(\perp) \leq f(f(\perp)) \leq f(f(f(\perp))) \leq \cdots$$

Since chains have lubs we have

$$x = \bigsqcup \{\perp, f(\perp), f(f(\perp)), f(f(f(\perp))), \ldots\}.$$

Now use continuity to show that x is a fixed point of f and the fact that $\perp$ is the least element to show that x is the *least* fixed point.

Science fiction?

- It is easy to check that $\Gamma_{b,c}$ is continuous, so it has a least fixed point.
- Thus, we have completed the so-called denotational semantics of our little language.
- But do we believe it?
- The operational semantics was very plausible but why should the fixed-point semantics jive with how our intuitions about repeated executions work?

Relating the two semantics

$$(a, \sigma) \longrightarrow n \text{ if and only if } [\![a]\!](\sigma) = n.$$

$$(be, \sigma) \longrightarrow b \text{ if and only if } [\![be]\!](\sigma) = b.$$

$$(c, \sigma) \longrightarrow \sigma' \text{ if and only if } [\![c]\!](\sigma) = \sigma'.$$

The first two are easy to prove by structural induction. From right to left is easy to prove for the third by structural induction. For the other direction one has to start with a structural induction but nested inside it is an induction on the number of computational steps.

Motivation for λ-calculus

- We have succeeded in giving a compositional semantics for a simple (but Turing complete) programming language.
- Programs denote functions and combinations of program fragments (using ;) are modelled by function composition.
- This language, however, does not have any of the *abstraction* mechanisms that programmers need to structure larger programs.
- We need to package program pieces in ways that capture some functionality and then combine these pieces *without having to look inside*: this is what λ-abstraction does.

λ-calculus

- Syntax: Countable set of variables: $x, y, z, u, v, \ldots$ and terms are defined inductively:

  $$x | MN | \lambda x.M$$

  where M, N are meta-variables standing for arbitrary λ-calculus terms.
- This just has the ability to form functions and to apply them.
- Nevertheless it is already Turing complete.

Free and Bound Variables

- Let $FV(M)$ stand for the set of free variables in M.
- We define $FV(M)$ by induction on M
- $FV(x) = \{x\}$,
- $FV(MN) = FV(M) \cup FV(N)$ and
- $FV(\lambda x.M) = FV(M) \setminus \{x\}$.
- Variables that are not free are *bound*; they can be renamed at will.
- Bound variables are only there to indicate connections or links.

Substitution

$M[N/x]$: substitute N for *free* occurrences of x in M.

$$\begin{array}{lcl} x[N/x] & = & N \\ M_1M_2[N/x] & = & M_1[N/x]M_2[N/x] \\ (\lambda y.M)[N/x] & = & \lambda y.M[N/x] \end{array}$$

provided $x \neq y$ and y does not occur free in N.
If these things are the case then rename bound variables.

β-reduction

- The main computational “engine” of the λ-calculus

$$(\lambda x.M)N \longrightarrow M[N/x].$$

- Assume variables are renamed as needed to avoid name clashes.
- β-reduction steps can be performed inside terms;
- so a given term may have many opportunities and
- there is no specified order in which these reductions should be done.
- Matches one’s operational intuition of what it means to apply a function to an argument: substitute the argument for the paramemter in the body of the definition.

Rules for reduction: semi-formal

- $M \longrightarrow M'$ where M' is the same as M except for renaming some bound variables.
- $M \underset{\beta}{\longrightarrow} M'$ implies $M \longrightarrow M'$.
- $\longrightarrow$ is reflexively and transitively closed.
- $\dfrac{M \longrightarrow M'}{\lambda x.M \longrightarrow \lambda x.M'}$
- $\dfrac{M \longrightarrow M'}{MN \longrightarrow M'N}$
- $\dfrac{N \longrightarrow N'}{MN \longrightarrow MN'}$
- $\lambda x.Mx \longrightarrow M$ provided $x \notin FV(M)$.

Equality

- Equality is the least equivalence relation including reduction.
- One often writes $\lambda \vdash M = N$ for provable equality.
- $M = N$ does *not* mean that M reduces to N or the other way around.
- Perhaps M reduces to M_1 and N also reduces to M_1.
- In general, $M = N$ means that there is a finite sequence of terms $M_1, \ldots, M_k$ with $M \longrightarrow M_1$, $M_2 \longrightarrow M_1$, $M_2 \longrightarrow M_3$, $M_4 \longrightarrow M_3$ and so on ending with $N \longrightarrow M_k$.

Church-Rosser Theorem

- Say a term has a *normal form* if there are no places to perform reduction; it has reached its final value.
- Church-Rosser: A term can have at most one normal form.
- This follows from the *diamond lemma*: if $M \xrightarrow{*} M_1$ and $M \xrightarrow{*} N_2$ then there is some M_3 such that $M_1 \xrightarrow{*} M_3$ and $M_2 \xrightarrow{*} M_3$.
- One is free to choose strategies to optimize the number of steps needed to reach normal form.
- The search for optimal reduction strategies was finally settled in 1990 (Abadi-Gonthier-Levy and Lamping) with a lot of illumination coming from the proof theory of linear logic (Girard, Danos-Regnier, Lafont).

Models of the λ-calculus

- The λ-calculus is meant to formalize the notion of function formation and application.
- The official definition captures the dynamical aspects of the formalism: how terms change as reduction occurs.
- What about the “static” view of functions as described in set theory? $f : A \longrightarrow B$.
- Functions from where to where?
- Functions from some set D to *itself*? So $D \longrightarrow D$.
- Any lambda term can be applied to anything *including itself*.
- Thus $D \simeq [D \longrightarrow D]$!!
- We can easily construct an example of this in plain old set theory.
- Take D to be a one-point set.
- Unfortunately (actually fortunately) this is the *only* example!

Denotational semantics of the λ-calculus I

Assume we have found a set $D \simeq [D \longrightarrow D]$.

Let $\eta : D \longrightarrow [D \longrightarrow D]$ and $\psi : [D \longrightarrow D] \longrightarrow D$ define the isomorphism.

Define an *environment* as a map $\rho : \mathrm{Var} \longrightarrow D$.

Update: $\rho[x \mapsto d]$: new environment like ρ *except* for the new binding explicitly shown.

Meaning function: $[\![\cdot]\!] : \text{Terms} \longrightarrow (\text{Env} \longrightarrow D)$.

The Semantic Equations

$[\![x]\!]\rho = \rho(x)$
$[\![MN]\!]\rho = \eta([\![M]\!]\rho)([\![N]\!]\rho)$
$[\![\lambda x.M]\!] = \psi(d \mapsto [\![M]\!]\rho[x \mapsto d])$.

Denotational Semantics of λ-calculus II

- Clearly $[D \longrightarrow D]$ cannot be *all* set theoretic functions from D to D.
- So how do we know that the functions $d \mapsto ([\![M]\!]\rho[x \mapsto d])$ are included in $[D \longrightarrow D]$?
- Albert Meyer rightly complained that he was never given a *definition* of a model of the λ-calculus, just examples.
- In a beautiful paper called “What is a model of the λ-calculus?” he gave an algebraic definition of what exactly is a model of the λ-calculus.
- I will not give this here, but will note that any of the examples are indeed lambda models.

Fixed-point combinators

- Consider $Y = (\lambda f.\lambda x.f(xx))(\lambda f.\lambda x.f(xx))$.
- $\lambda \vdash YM = M(YM)$.
- Thus Y finds fixed points of any term.
- Thus whatever D we have; it better support some pretty powerful fixed-point theory.

What can one express in the λ-calculus?

- One can express booleans and conditionals: indeed the booleans *are* the conditionals.
- One can express arithmetic using several different numeral systems.
- One can express recursion with the Y combinator.
- In short, the λ-calculus, *without any added structures for arithmetic*, is already Turing complete.

Bad behaviour

- The term $\Delta\Delta$ where $\Delta = \lambda x.xx$ has no formal form: it reduces to itself.
- The term $(\lambda x.\lambda y.y)(\Delta\Delta)(\lambda u.u)$ has a normal form, but if you choose the wrong order to do reductions it might go into a loop.
- Note this is not prevented by the Church-Rosser theorem.
- Of course, one expects non-termination with a Turing-complete language.
- If a term has a normal form we say that it is *normalizing*.
- If every reduction sequence terminates we say that it is *strongly normalizing*.
- Wouldn't it be nice if we could "tame" the λ-calculus and guarantee that every term has a normal form?

Yes we can! Simply typed lambda calculus

- We have a ground type; call it $\mathbf{0}$.
- If τ_1 and τ_2 are types so is $\tau_1 \longrightarrow \tau_2$.
- Variables come labelled with types: x^τ or $x : \tau$.
- Typing judgements have the form $\Gamma \vdash M : \tau$; where Γ is a list of variable typing assumptions.
- $\dfrac{}{\Gamma, x : \tau \vdash x : \tau}$
- $\dfrac{\Gamma, x : \tau \vdash x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x.M : \tau \longrightarrow \tau'}$
- $\dfrac{\Gamma \vdash M : \tau \longrightarrow \tau' \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \tau'}$
- In this calculus; one cannot write terms like Δ. One can prove that *every* term is *strongly normalizing*.
- One can also show that the expressiveness is *severely* limited!!

PCF

- The simply-typed λ-calculus is well-behaved but very weak.
- Perhaps we should put recursion back in a controlled way: PCF defined by Plotkin in 1977.
- Essentially simply-typed λ-calculus with two ground types: integers and booleans and some basic logical and arithmetic operations plus
- recursion at every type.
- Of course not every term will have a normal form.

Syntax of PCF

- Types
$$\tau ::== Nat | Bool | \tau_1 \times \tau_2 | \tau_1 \longrightarrow \tau_2.$$
- Ground Constants
$$tt, ff \ : \ Bool, \quad 0, 1, 2 \ldots n, \ldots : \ Nat$$
- Higher-type constants
$$plus : Nat \times Nat \longrightarrow Nat, \quad equal : Nat \times Nat \longrightarrow Bool,$$
$$if_\tau : Bool \times \tau \times \tau \longrightarrow \tau, \quad fix_\tau : (\tau \longrightarrow \tau) \longrightarrow \tau.$$
- Products $\dfrac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash (M,\ N) : \sigma \times \tau}$ $\dfrac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \pi_{1(2)}(M) : \tau_{1(2)}}$
- Fixed points $\dfrac{\Gamma, x : \tau \vdash M : \tau}{\Gamma \vdash fix\ x.M : \tau}$
- Other terms are as in the simply-typed λ-calculus.

Operational semantics of PCF

- Constants evaluate to themselves $\dfrac{}{1729 \longrightarrow 1729}$
- Algebraic expressions follow the obvious rules, e.g.
  $\dfrac{M_1 \longrightarrow n_1 \quad M_2 \longrightarrow n_2}{plus(M_1, M_2) \longrightarrow (n_1 + n_2)}$
- Rules for fixed points

  $\dfrac{}{fix\ x.M \longrightarrow M[fix\ x.M/x]}$
- Other rules are what you expect.

Towards a denotational semantics for PCF

- We can model the base types easily, except that we have to have a special value to denote expressions that may not terminate.
- How do we model recursion? With fixed-point theory of course!
- We have a simple example of a poset where fixed points for continuous functions exist
- but what about those function types?
- We need a theory of “domains” that guarantees us the appropriate function spaces have fixed points.
- We need ways of arguing that the denotational semantics and the operational semantics “match up well” (whatever that means!).