Semantics: The goal of semantics is to define the meaning of a programming language. Prose is a bad way to do this; it is imprecise, subjective, impossible to analyze.

Semantics is a collection of techniques which define meaning of a programming language formally.

Operational Methods: *Operational* methods define a *term rewriting system* (TRS) which rewrites terms syntactically. Operational semantics consists of a set of rules, typically written in the form $t \to t'$. Example:

$$\texttt{if true } t_t \texttt{ else } t_f \to t_t$$

This can be seen as describing an *interpreter* for the language.

Type Systems: Type systems in operational models typically take the form of logical inference (aka. Gentzen-style) rules about the syntax. Safety is proven by showing that every well-typed term can be evaluated one step, and evaluating a well-typed term yields a well-typed term. These proofs can get quite complicated.

Problems: Operational semantics is a good approach for many applications; however, it has problems:

- It describes the language in terms of a low-level description of an inefficient implementation
- Operational models can be hard to get right, and harder to analyze.
- Inductive proofs can get nasty when combining multiple "directions" of induction (general recursion, infinitely-deep heaps, infinite-rank types, arbitrarily long executions)
- Operations methods have trouble with infinite or infinite-rank objects (rank-ω types, for example)

Denotational Methods: *Denotational* methods define the meaning of a language by mapping each term to a mathematical object. For example, we would define a mapping that would take any implementation of the factorial function to the mathematical definition of factorial.

Denotational methods have some advantages over operational methods; they also have some disadvantages:

- Denotational methods can directly describe infinite and infinite-rank objects without much trouble
- Multiple "dimensions" of induction are much easier to accomplish
- Easier to analyze the language mathematically
- Less rigid, low-level definition of behavior
- On the other hand, it tends to be much harder to define denotational semantics for an existing language
- Denotational methods are less well-known and can involve much more and much harder work to construct if describing new features

Denotational methods are also very useful for addressing other problems: constraint systems, abstract interpretation, others.

Overview of Methods: In denotational methods, we define a structured space, or *domain* which contains all possible values in the language. We model datatype constructors using *constructions* on spaces (ex. cartesian products, disjoint unions, function spaces).

There are two popular approaches. Dana Scott's original approach was based on lattices (partial orders). Approaches based on metric spaces emerged later. The two approaches share many similar ideas.

Partial Orders: A *partial order* is a set with an order relation $\sqsubset$. Unlike a total order, some elements may be incomparable. We often name the $\sqsubset$ relation "below". The following are common conventions for partial orders:

- A *join*, *least upper bound*, or *supremum* $\bigsqcup_i a_i$ of elements a_i is an element s such that $a_i \sqsubseteq s$ for all i, and there is no element s_0 such that $s_0 \sqsubseteq s$. This is the smallest element above (or equal to) all a_i.

- A *meet*, *greatest lower bound*, or *infimum* $\bigsqcap_i a_i$ is the dual notion (the largest element below (or equal to) all a_i)
- A *join-semilattice* is a partial order in which every set of elements has a join. A *meet-semilattice* is a partial order in which ever set of elements has a meet.
- The lowest element in a partial order is typically called "bottom", and represented with the symbol $\bot$.

Order of Approximation: In Scott-style semantics, the order represents a notion of *definiteness* or *approximation*, and $\bot$ represents a notion of "undefined". For example, the expression $1 + 2$ is more defined than $\bot + 2$ or $1 + \bot$ (in fact, it is the join of these two expressions).

As we move up the partial order, an expression becomes increasingly more defined. When the expression is fully-defined, we are at a *fixed point*. This sort of refinement can be characterized as a *(Scott-)continuous monotone function*. Such functions have the following properties:

- *Monotone*: $x \sqsubseteq f(x)$, the result is always above (or equal) to the argument.
- *(Scott-)Continuous*: $f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$, f preserves joins.

To see an example of this, consider refining the denotation of a factorial implementation. We have a definition for 0!, and we have a definition for $n!$ in terms of $(n-1)!$: $n! = n \times (n-1)!$. Let $!_k$ be an approximation defined for $[0, k]$. We have the folloing sequence:

$$!_0 = \{0 \mapsto 1, n > 0 \mapsto \bot\} \sqsubset$$

$$!_1 = \{0 \mapsto 1, 1 \mapsto 1, n > 1 \mapsto \bot\} \sqsubset$$

$$!_2 = \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, n > 2 \mapsto \bot\} \sqsubset$$

$$!_3 = \{0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 6, n > 3 \mapsto \bot\} \sqsubset$$

and so on up to $!_k$. The *limit* $!_\omega$ of this sequence *is* the full factorial function.

Scott Domains: We will want to build a space of values to which we will map every term in the language. To do this, we use a special class of spaces (called "domains"), and we represent datatype constructors by *constructions* on domains.

There are two characterizations of Scott-domains, which are equivalent to one another. One is *chain-complete partial orders* (CCPO), which is a partial order with the guarantee that any *ω-chain* (any sequence $a_0 \sqsubseteq a_1 \sqsubseteq \ldots \sqsubseteq a_\omega$ has a join. The second, more commonly used is *directed-complete partial orders* (DCPO), which is a partial order where any *directed subset* (any subset S such that for any $a, b \in S$, there exists $c \sqsupseteq a, b$) has a join. Note that these properties are somewhat obvious for finite sets; it is the infinite case that we are concerned about.

"Scott-domain" generally refers to a DCPO/CCPO.

In any CCPO, any monotone continuous function defines an chain and therefore, has a limit.

Constructions: There are several useful constructions on Scott-domains, which yield spaces that are again Scott-domains (let A and B be two domains in the following list):

- *Cartesian Product*: $A \times B$, the set of all pairs (a, b) where $a \in A$ and $b \in B$. This can also be written as $\prod_i A_i$
- *Smash Product*: $A \otimes B$, same as a cartesian product, but with $\bot \otimes b = a \otimes \bot = \bot$. This can also be written as $\bigotimes_i A_i$
- *Disjoint Union*: $A \cup B$ (often written $A \oplus B$), where A and B are treated as disjoint sets. This can also be written as $\bigoplus_i A_i$
- *Pointed*: $A_\bot$, Add a bottom element to A (this is a disjoint union with a one-element set)
- *(Continuous) Function Space*: $[A \to B]$, the space of continuous functions from A to B.

- *Strict Continuous Function Space*: $[A \to_! B]$, the space of continuous functions from A to B such that $f(\bot) = \bot$

If A and B are both DCPO's in these constructions, then the result is again a DCPO.

Some of these correspond to classic datatypes:

- Cartesian products (and smash products) correspond to tuples.
- Cartesian powers correspond to arrays.
- Function spaces correspond to functions.
- The "strict" (smash products, strict functions spaces) correspond to strict datatypes.

The disjoint union construction is used to "glue" multiple spaces together. The pointed construction is used to add a bottom value if one does not exist.

Retracts: For any of the previous constructions $F(A, B)$ on A and B, we can build an *embedding/projection pair* (e_A, p_A) of A (and B) in the resulting space, such that $p_A(e_A(a)) = a$ for all $a \in A$ (and the same for B). Note that $p_A \circ e_A = \mathrm{id}_A$. It is necessarily not an isomorphism, though, since p_A takes a larger space to a smaller one, thus $e_A \circ p_A$ is not necessarily $\mathrm{id}_{F(A,B)}$

A space A is a *retract* of a space B if there is an embedding/projection pair from A to B that preserves the original order. That is, $e_A(a_1) \sqsubset e(a_2)$ iff $a_1 \sqsubset a_2$.

Retracts compose nicely. If A is a retract of B by (e_1, p_2) and B is a retract of C by (e_2, p_2), then A is a retract of C by $(e_2 \circ e_1, p_2 \circ p_1)$. In this capacity, the embedding/projection pairs act similarly to continuous monotone functions.

A space can be a retract of itself. In this case, the embedding/projection pair *are* monotone continuous functions.

For all of the above constructions, we form retracts:

- For products spaces, $e_A(a) = (a, 1_B)$ and $p_A(a, 1_B) = a$ for some distinguished element $1_B \in B$ (often a "unit" value)
- For disjoint unions (and pointed sets), the embedding/projection pair is just the identity map.
- For function spaces, $e_A(a) = \lambda x.a$ and $p_a(f) = f(1_A)$ (again, 1_A is some distinguished element of A)

Recursive Domain Equations: To describe infinite and infinite-rank objects, we build a space described by a *recursive domain equation*. The following example describes λ-calculus:

$$D \cong [D \to D]$$

or more generally,

$$D \cong F(D), \text{where } F(D) = [D \to D]$$

That is, D is isomorphic to its own continuous function space.

If D is a retract of $F(D)$, then the embedding/projection pairs are isomorphisms (and therefore, continuous monotone functions).

Limit (Fixed-Point) Spaces: For some recursive domain equation $D \cong F(D)$ where D is a retract of $F(D)$, we can characterize D as the *limit* of the sequence $D_0, D_1 = F(D_0), D_2 = F^2(D_0), \ldots, D_\omega = F^\omega(D_0)$, where D_0 is a starting set (often called the *basis*).

We have a continuous monotone map between any D_i, D_j. Thus, we can characterize any point in D_ω as a fixed-point (alterantively, the limit of a sequence).

The *Knaster-Tarski fixed-point* theorem shows that the space of fixed points of a continuous map applied to a DCPO is itself a DCPO. However, we cannot use this directly for this construction, as the spaces D_i get progressively bigger.

Inverse Limits: We would like to describe D_ω as the space of limits of the sequences defined by repeatedly applying the embeddings e_i. However, this doesn't work, since e_is are not surjective. Instead, we must rely on a dual concept of *inverse* limits.

Suppose we have a sequence of DCPO's D_i, p_i such that D_i is a retract of D_{i+1} (and therefore, all D_{i+k}. We can build a chain out of the projections as follows: make every p_i send exactly one value of D_i to D_{i-1}, and everything else to $\perp$. We can then define $p_{i,j} = p_i \circ ... \circ p_j$ where $i < j$, such that $D_i = p_{i+1,j}(D_j)$. This does form a chain, so we can define $p_{i,\omega} = \lim_{j \to \omega} p_{i,j}$.

The *inverse limit* D_ω of the sequence D_i is a domain such that for any i, $D_i = p_{i,\omega}(D_\omega)$. Scott's inverse limit theorem shows that this space exists and is unique, that the p_i's uniquely determine embedding functions e_i, and that the inverse limit is also the direct limit.

Categorical Constructions: Since Scott's work, the domain-theoretic constructions have become increasingly categorical.

Limit language is inverted in category theory: Categorical *limits* correspond to Scott's inverse limits; categorical *colimits* correspond to direct limits.

Main challenge in categorical constructions is contravariance. Smyth and Plotkin use a category where arrows are retracts. Freyd splits into covariant and contravariant parts and uses a construction similar to how we build general logic functions with monotone logic functions.

Metric Spaces: Metric spaces can also form a basis for denotational semantics. This has several advantages:

- Closer to classical mathematics (analysis).
- Tends to work better for representing heaps and stateful computation.
- No need for inverse limit theorem.

Metric spaces are a space equipped with a distance function $d(x, y)$ such that $d(x, y) = d(y, x)$, $d(x, y) > 0$ if $x \neq y$, $d(x, x) = 0$ and $d(x, z) \leq d(x, y) + d(y, z)$.

A "ϵ-ball" (around a center point) in a metric space is a set containing all points whose distance from a center point is at most ϵ.

The *limit* of a sequence in a metric space is a point such that for any ϵ, there is a point in the sequence such that no subsequent point's distance from the limit never exceeds ϵ.

In metric spaces, some incomplete term is a ball, representing the possible values. As we refine, we shrink the ball further and further until we have a single point. The analogue of $\perp$ is the entire space.

Metric Space Constructions: We have analogues of many of the same constructions that we used in the DCPO approach: cartesian products, continuous function spaces, disjoint unions (often called "direct sums")

Recursive Metric Space Equations: Banach's fixed-point theorem gives us a general construction for recursive metric space equations, based on *contractive maps*. It assumes a complete, non-empty metric space (note that any metric space can be turned into a complete metric space).

A contractive map F on a space X guarantees that $d(F(x), F(y)) < d(x, y)$.

Ultrametric Spaces: An *ultrametric space* is a metric space where the triangle inequality is replaced by $d(x, z) \leq \max[d(x, y), d(y, z)]$. An example is the *Cantor metric*, which defines the distance between two strings as 2^i, where i is the first character where they differ.

In a 1-bounded ultrametric space, all the common constructions are *non-expansive* maps, meaning $d(F(x), F(y)) \leq d(x, y)$ (almost but not quite contractive). Fortunately, the map $\frac{1}{2}$, which halfs all the distances is well-behaved, and we can use it to turn any non-expansive map into a contractive one.

Thus, in complete, 1-bounded, non-empty ultrametric spaces, we can solve recursive space equations.